2.5 Input and Output

Most interesting programs ask the program user to provide input values and produce outputs that depend on the user input. In the following sections, you will see how to read user input and how to control the appearance of the output that your programs produce.

2.5.1 User Input

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Consider, for example, the `initials.py` program from Section 2.4.4 that prints a pair of initials. The two names from which the initials are derived are specified as literal values. If the program user entered the names as inputs, the program could be used for any pair of names.

When a program asks for user input, it should first print a message that tells the user which input is expected. Such a message is called a **prompt**. In Python, displaying a prompt and reading the keyboard input is combined in one operation.

```
first = input("Enter your first name: ")
```

Use the `input` function to read keyboard input.

The `input` function displays the string argument in the console window and places the cursor on the same line, immediately following the string.

```
Enter your first name: 
```

Note the space between the colon and the cursor. This is common practice in order to visually separate the prompt from the input. After the prompt is displayed, the program waits until the user types a name. After the user supplies the input,

```
Enter your first name: Rodolfo
```

the user presses the Enter key. Then the sequence of characters is returned from the `input` function as a string. In our example, we store the string in the variable `first` so it can be used later. The program then continues with the next statement.

The following version of the `initials.py` program is changed to obtain the two names from the user.

ch02/initials2.py

```
1  ##
2  #  This program obtains two names from the user and prints a pair of initials.
3  #
4
5  # Obtain the two names from the user.
6  first = input("Enter your first name: ")
7  second = input("Enter your significant other's first name: ")
8
9  # Compute and display the initials.
10 initials = first[0] + "&" + second[0]
11 print(initials)
```

Program Run

```
Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S
```

2.5.2 Numerical Input

The `input` function can only obtain a string of text from the user. But what if we need to obtain a numerical value? Consider, for example, a program that asks for the price and quantity of soda containers. To compute the total price, the number of soda containers needs to be an integer value, and the price per container needs to be a floating-point value.

To read an integer value, first use the `input` function to obtain the data as a string, then convert it to an integer using the `int` function.

To read an integer or floating-point value, use the `input` function followed by the `int` or `float` function.

```
userInput = input("Please enter the number of bottles: ")
bottles = int(userInput)
```

In this example, `userInput` is a temporary variable that is used to store the string representation of the integer value (see Figure 5). After the input string is converted to an integer value and stored in `bottles`, it is no longer needed.

To read a floating-point value from the user, the same approach is used, except the input string has to be converted to a `float`.

```
userInput = input("Enter price per bottle: ")
price = float(userInput)
```

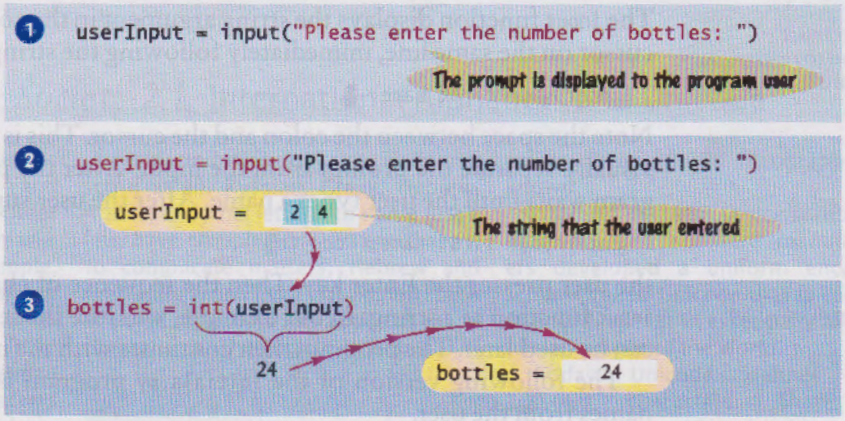


Figure 5
Extracting an Integer Value

2.5.3 Formatted Output

When you print the result of a computation, you often want to control its appearance. For example, when you print an amount in dollars and cents, you usually want it to be rounded to two significant digits. That is, you want the output to look like

```
Price per liter: 1.22
```

instead of

```
Price per liter: 1.215962441314554
```

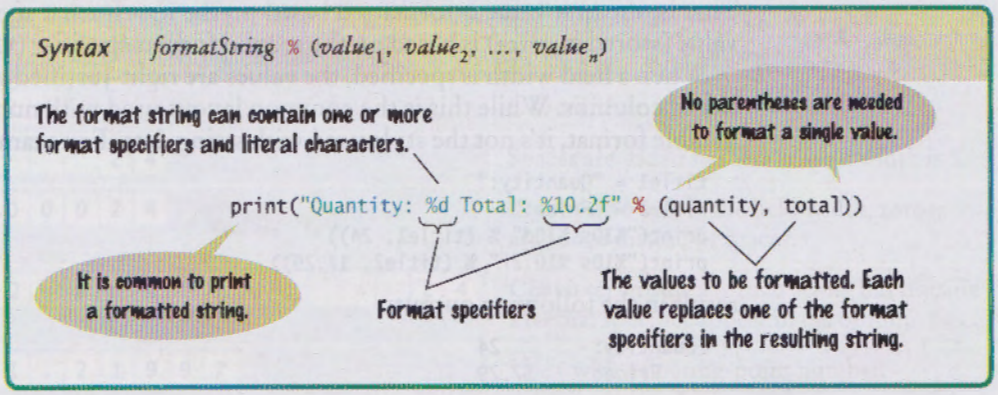
The following command displays the price with two digits after the decimal point:

```
print("%.2f" % price)   # Prints 1.22
```

You can also specify a field width (the total number of characters, including spaces), like this:

```
print("%10.2f" % price)
```

Syntax 2.3 String Format Operator



The price is printed right-justified using ten characters: six spaces followed by the four characters 1.22.

```
      1.22
```

The argument passed to the `print` function

```
"%10.2f" % price
```

specifies how the string is to be formatted. The result is a string that can be printed or stored in a variable.

Use the string format operator to specify how values should be formatted.

You learned earlier that the `%` symbol is used to compute the remainder of floor division, but that is only the case when the values left and right of the operator are both numbers. If the value on the left is a string, then the `%` symbol becomes the **string format operator**.

The construct `%10.2f` is called a *format specifier*: it describes how a value should be formatted. The letter `f` at the end of the format specifier indicates that we are formatting a floating-point value. Use `d` for an integer value and `s` for a string; see Table 9 on page 59 for examples.

The *format string* (the string on the left side of the string format operator) can contain one or more format specifiers and literal characters. Any characters that are not format specifiers are included verbatim. For example, the command

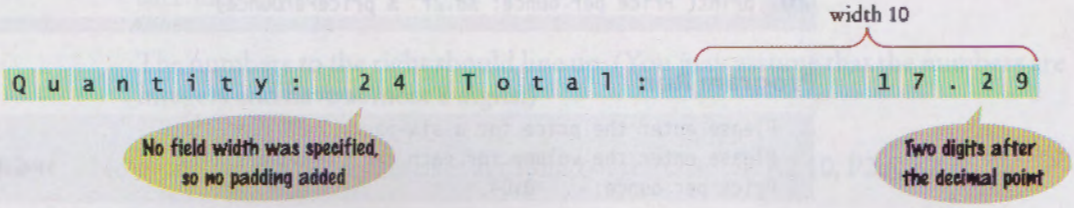
```
"Price per liter:%10.2f" % price
```

produces the string

```
"Price per liter:      1.22"
```

You can format multiple values with a single string format operation, but you must enclose them in parentheses and separate them by commas. Here is a typical example:

```
print("Quantity: %d Total: %10.2f" % (quantity, total))
```



The values to be formatted (`quantity` and `total` in this case) are used in the order listed. That is, the first value is formatted based on the first format specifier (`%d`), the second value (stored in `price`) is based on the second format specifier (`%10.2f`), and so on.

When a field width is specified, the values are right-justified within the given number of columns. While this is the common layout used with numerical values printed in table format, it's not the style used with string data. For example, the statements

```
title1 = "Quantity:"
title2 = "Price:"
print("%10s %10d" % (title1, 24))
print("%10s %10.2f" % (title2, 17.29))
```

result in the following output:

```
 Quantity:         24
    Price:      17.29
```

The output would look nicer, however, if the titles were left-justified. To specify left justification, add a minus sign before the string field width:

```
print("%-10s %10d" % (title1, 24))
print("%-10s %10.2f" % (title2, 17.29))
```

The result is the far more pleasant

```
Quantity:         24
Price:         17.29
```

Our next example program will prompt for the price of a six-pack and the volume of each can, then print out the price per ounce. The program puts to work what you just learned about reading input and formatting output.

ch02/volume2.py

```
1  ##
2  #  This program prints the price per ounce for a six-pack of cans.
3  #
4
5  # Define constant for pack size.
6  CANS_PER_PACK = 6
7
8  # Obtain price per pack and can volume.
9  userInput = input("Please enter the price for a six-pack: ")
10 packPrice = float(userInput)
11
12 userInput = input("Please enter the volume for each can (in ounces): ")
13 canVolume = float(userInput)
14
15 # Compute pack volume.
16 packVolume = canVolume * CANS_PER_PACK
17
18 # Compute and print price per ounce.
19 pricePerOunce = packPrice / packVolume
20 print("Price per ounce: %8.2f" % pricePerOunce)
```

Program Run

```
Please enter the price for a six-pack: 2.95
Please enter the volume for each can (in ounces): 12
Price per ounce:     0.04
```

Table 9 Format Specifier Examples

Format String	Sample Output	Comments
`"%d"`	`24`	Use `d` with an integer.
`"%5d"`	`␣␣␣24`	Spaces are added so that the field width is 5.
`"%05d"`	`00024`	If you add `0` before the field width, zeroes are added instead of spaces.
`"Quantity:%5d"`	`Quantity:␣␣␣24`	Characters inside a format string but outside a format specifier appear in the output.
`"%f"`	`1.219997`	Use `f` with a floating-point number.
`"%.2f"`	`1.22`	Prints two digits after the decimal point.
`"%7.2f"`	`␣␣␣1.22`	Spaces are added so that the field width is 7.
`"%s"`	`Hello`	Use `s` with a string.
`"%d %.2f"`	`24 1.22`	You can format multiple valnes at once.
`"%9s"`	`␣␣␣␣Hello`	Strings are right-justified by default.
`"%-9s"`	`Hello␣␣␣␣`	Use a negative field width to left-justify.
`"%d%%"`	`24%`	To add a percent sign to the output, use `%%`.

22. Write statements to prompt for and read the user's age.

23. What is problematic about the following statement sequence?

```
userInput = input("Please enter the unit price: ")
unitPrice = int(userInput)
```

24. What is problematic about the following statement sequence?

```
userInput = input("Please enter the number of cans")
cans = int(userInput)
```

25. What is the output of the following statement sequence?

```
volume = 10
print("The volume is %5d" % volume)
```

26. Using the string format operator, print the values of the variables `bottles` and `cans` so that the output looks like this:

```
Bottles:      8
Cans:        24
```

The numbers to the right should line up. (You may assume that the numbers are integers and have at most 8 digits.)

Practice It Now you can try these exercises at the end of the chapter: R2.10, P2.6, P2.7.

Programming Tip 2.4

Don't Wait to Convert

When obtaining numerical values from input, you should convert the string representation to the corresponding numerical value immediately after the input operation.

Obtain the string and save it in a temporary variable that is then converted to a number by the next statement. Don't save the string representation and convert it to a numerical value every time it's needed in a computation:

```
unitPrice = input("Enter the unit price: ")
price1 = float(unitPrice)
price2 = 12 * float(unitPrice)    # Bad style
```

It is bad style to repeat the same computation multiple times. And if you wait, you could forget to perform the conversion.

Instead, convert the string input immediately to a number:

```
unitPriceInput = input("Enter the unit price: ")
unitPrice = float(unitPriceInput)    # Do this immediately after reading the input
price1 = unitPrice
price2 = 12 * unitPrice
```

Or, even better, combine the calls to `input` and `float` in a single statement:

```
unitPrice = float(input("Enter the unit price: "))
```

The string returned by the input function is passed directly to the `float` function, not saved in a variable.

HOW TO 2.1

Writing Simple Programs

This How To shows you how to turn a problem statement into pseudocode and, ultimately, a Python program.

Problem Statement Write a program that simulates a vending machine. A customer selects an item for purchase and inserts a bill into the vending machine. The vending machine dispenses the purchased item and gives change. Assume that all item prices are multiples of 25 cents, and the machine gives all change in dollar coins and quarters. Your task is to compute how many coins of each type to return.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there are two inputs:

- The denomination of the bill that the customer inserts
- The price of the purchased item

A vending machine takes bills and gives change in coins.

There are two desired outputs:

- The number of dollar coins that the machine returns
- The number of quarters that the machine returns

Step 2 Work out examples by hand.

This is a very important step. If you can't compute a couple of solutions by hand, it's unlikely that you'll be able to write a program that automates the computation.

Let's assume that a customer purchased an item that cost $2.25 and inserted a $5 bill. The customer is due $2.75, or two dollar coins and three quarters, in change.

That is easy for you to see, but how can a Python program come to the same conclusion? The key is to work in pennies, not dollars. The change due the customer is 275 pennies. Dividing by 100 yields 2, the number of dollars. Dividing the remainder (75) by 25 yields 3, the number of quarters.

Step 3 Write pseudocode for computing the answers.

In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general.

Given an arbitrary item price and payment, how can you compute the coins due? First, compute the change due in pennies:

```
change due = 100 x bill value - item price in pennies
```

To get the dollars, divide by 100 and discard the fractional part:

```
dollar coins = change due divided by 100 (without the fractional part)
```

If you prefer, you can use the Python symbol for floor division.

```
dollar coins = change due // 100
```

But you don't have to. The purpose of pseudocode is to describe the computation in a humanly readable form, not to use the syntax of a particular programming language.

The remaining change due can be computed in two ways. If you are aware that one can compute the remainder of a floor division (in Python, with the modulus operator), you can simply compute

```
change due = remainder of dividing change due by 100
```

Alternatively, subtract the penny value of the dollar coins from the change due:

```
change due = change due - 100 x dollar coins
```

To get the quarters due, divide by 25:

```
quarters = change due // 25
```

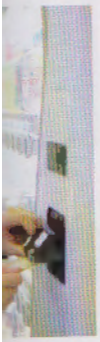
Step 4 Declare the variables and constants that you need, and decide what types of values they hold.

Here, we have five variables:

- `billValue`
- `itemPrice`
- `changeDue`
- `dollarCoins`
- `quarters`

Should we introduce constants to explain 100 and 25 as `PENNIES_PER_DOLLAR` and `PENNIES_PER_QUARTER`? Doing so will make it easier to convert the program to international markets, so we will take this step.

Because we use floor division and the modulus operator, we want all values to be integers.



Step 5 Turn the pseudocode into Python statements.

If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations (such as floor division and modulus) in Python.

```
changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice
dollarCoins = changeDue // PENNIES_PER_DOLLAR
changeDue = changeDue % PENNIES_PER_DOLLAR
quarters = changeDue // PENNIES_PER_QUARTER
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the bill value and item price:

```
userInput = input("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ")
billValue = int(userInput)
userInput = input("Enter item price in pennies: ")
itemPrice = int(userInput)
```

When the computation is finished, we display the result. For extra credit, we format the output strings to make sure that the output lines up neatly:

```
print("Dollar coins: %6d" % dollarCoins)
print("Quarters:     %6d" % quarters)
```

Step 7 Provide a Python program.

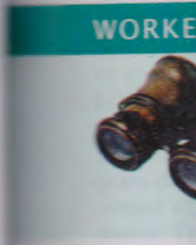
Your computation needs to be placed into a program. Find a name for the program that describes the purpose of the computation. In our example, we will choose the name `vending`.

In the program, you need to declare constants and variables (Step 4), carry out computations (Step 5), and provide input and output (Step 6). Clearly, you will want to first get the input, then do the computations, and finally show the output. Define the constants at the beginning of the program, and define each variable just before it is needed.

Here is the complete program:

ch02/vending.py

```
1  ##
2  #  This program simulates a vending machine that gives change.
3  #
4
5  # Define constants.
6  PENNIES_PER_DOLLAR = 100
7  PENNIES_PER_QUARTER = 25
8
9  # Obtain input from user.
10 userInput = input("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ")
11 billValue = int(userInput)
12 userInput = input("Enter item price in pennies: ")
13 itemPrice = int(userInput)
14
15 # Compute change due.
16 changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice
17 dollarCoins = changeDue // PENNIES_PER_DOLLAR
18 changeDue = changeDue % PENNIES_PER_DOLLAR
19 quarters = changeDue // PENNIES_PER_QUARTER
20
21 # Print change due.
22 print("Dollar coins: %6d" % dollarCoins)
23 print("Quarters:     %6d" % quarters)
```



Program Run

```
Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 225
Dollar coins:       2
Quarters:           3
```

WORKED EXAMPLE 2.2 Computing the Cost of Stamps

Problem Statement Simulate a postage stamp vending machine. A customer inserts dollar bills into the vending machine and then pushes a "purchase" button. The vending machine gives out as many first-class stamps as the customer paid for, and returns the change in penny (one-cent) stamps. A first-class stamp cost 44 cents at the time this book was written.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there is one input:

- The amount of money the customer inserts

There are two desired outputs:

- The number of first-class stamps the machine returns
- The number of penny stamps the machine returns

Step 2 Work out examples by hand.

Let's assume that a first-class stamp costs 44 cents and the customer inserts $1.00. That's enough for two stamps (88 cents) but not enough for three stamps ($1.32). Therefore, the machine returns two first-class stamps and 12 penny stamps.

Step 3 Write pseudocode for computing the answers.

Given an amount of money and the price of a first-class stamp, how can you compute how many first-class stamps can be purchased with the money? Clearly, the answer is related to the quotient

$$\frac{\text{amount of money}}{\text{price of first-class stamp}}$$

For example, suppose the customer paid $1.00. Use a pocket calculator to compute the quotient: $1.00/$0.44 = 2.27.

How do you get "2 stamps" out of 2.27? It's the quotient without the fractional part. In Python, this is easy to compute if both arguments are integers. Therefore, let's switch our computation to pennies. Then we have

```
number of first-class stamps = 100 / 44 (without remainder)
```

What if the user inputs two dollars? Then the numerator becomes 200. What if the price of a stamp goes up? A more general equation is

```
number of first-class stamps = 100 x dollars / price of first-class stamps in cents (without remainder)
```

How about the change? Here is one way of computing it. When the customer gets the stamps, the change is the customer payment, reduced by the value of the stamps purchased. In our example, the change is 12 cents—the difference between 100 and 2 · 44. Here is the general formula:

```
change = 100 x dollars – number of first-class stamps x price of first-class stamp
```

Step 4 Define the variables and constants that you need, and decide what types of values they hold.

Here, we have three variables:

- `dollars`
- `firstClassStamps`
- `change`

There is one constant, `FIRST_CLASS_STAMP_PRICE`.

The variable `dollars` and constant `FIRST_CLASS_STAMP_PRICE` must be integers because the computation of `firstClassStamps` uses floor division. The remaining variables are also integers, counting the number of first-class and penny stamps.

Step 5 Turn the pseudocode into Python statements.

Our computation depends on the number of dollars that the user provides. Translating the math into Python yields the following statements:

```
firstClassStamps = 100 * dollars // FIRST_CLASS_STAMP_PRICE
change = 100 * dollars - firstClassStamps * FIRST_CLASS_STAMP_PRICE
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the number of dollars and obtain the value:

```
dollarStr = input("Enter number of dollars: ")
dollars = int(dollarStr)
```

When the computation is finished, we display the result.

```
print("First class stamps: %6d" % firstClassStamps)
print("Penny stamps:       %6d" % change)
```

Step 7 Write a Python program.

Here is the complete program:

ch02/stamps.py

```
1  ##
2  #  This program simulates a stamp machine that receives dollar bills and
3  #  dispenses first class and penny stamps.
4  #
5
6  # Define the price of a stamp in pennies.
7  FIRST_CLASS_STAMP_PRICE = 44
8
9  # Obtain the number of dollars.
10 dollarStr = input("Enter number of dollars: ")
11 dollars = int(dollarStr)
12
13 # Compute and print the number of stamps to dispense.
14 firstClassStamps = 100 * dollars // FIRST_CLASS_STAMP_PRICE
15 change = 100 * dollars - firstClassStamps * FIRST_CLASS_STAMP_PRICE
16 print("First class stamps: %6d" % firstClassStamps)
17 print("Penny stamps:       %6d" % change)
```

Program Run

```
Enter number of dollars: 4
First class stamps:      9
Penny stamps:            4
```

Computing & Society 2.2 The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the Pentium. Unlike previous generations of its processors, it had a very fast floating-point unit. Intel's goal was to compete aggressively with the makers of higher-end processors for engineering workstations. The Pentium was a huge success immediately.

In the summer of 1994, Dr. Thomas Nicely of Lynchburg College in Virginia ran an extensive set of computations to analyze the sums of reciprocals of certain sequences of prime numbers. The results were not always what his theory predicted, even after he took into account the inevitable roundoff errors. Then Dr. Nicely noted that the same program did produce the correct results when running on the slower 486 processor that preceded the Pentium in Intel's lineup. This should not have happened. The optimal roundoff behavior of floating-point calculations has been standardized by the Institute for Electrical and Electronic Engineers (IEEE) and Intel claimed to adhere to the IEEE standard in both the 486 and the Pentium processors. Upon further checking, Dr. Nicely discovered that indeed there was a very small set of numbers for which the product of two numbers was computed differently on the two processors. For example,

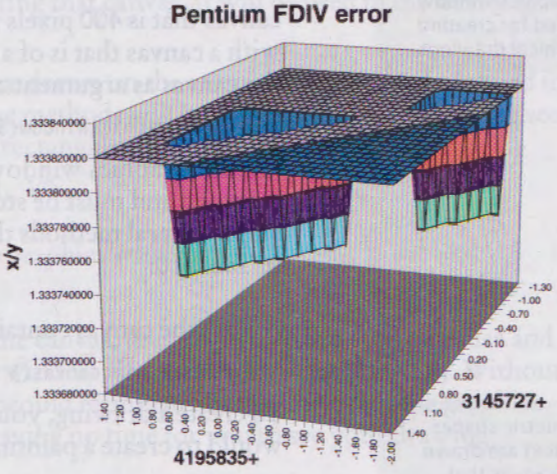
$$4{,}195{,}835 - ((4{,}195{,}835/3{,}145{,}727) \times 3{,}145{,}727)$$

is mathematically equal to 0, and it did compute as 0 on a 486 processor. On his Pentium processor the result was 256.

As it turned out, Intel had independently discovered the bug in its testing and had started to produce chips that fixed it. The bug was caused by an error in a table that was used to speed up the floating-point multiplication algorithm of the processor. Intel determined that the problem was exceedingly rare. They claimed that under normal use, a typical consumer would only notice the problem once every 27,000 years. Unfortunately for Intel, Dr. Nicely had not been a normal user.

Now Intel had a real problem on its hands. It figured that the cost of replacing all Pentium processors that it had sold so far would cost a great deal of money. Intel already had more orders for the chip than it could produce, and it would be particularly galling to have to give out the scarce chips as free replacements instead of selling them. Intel's management decided to punt on the issue and initially offered to replace the processors only for those customers who could prove that their work required absolute precision in mathematical calculations. Naturally, that did not go over well with the hundreds of thousands of customers who had paid retail prices of $700 and more for a Pentium chip and did not want to live with the nagging feeling that perhaps, one day, their income tax program would produce a faulty return.

Ultimately, Intel caved in to public demand and replaced all defective chips, at a cost of about 475 million dollars.



This graph shows a set of numbers for which the original Pentium processor obtained the wrong quotient.

2.6 Graphics: Simple Drawings

There are times when you may want to include simple drawings such as figures, graphs, or charts in your programs. Although the Python library provides a module for creating full graphical applications, it is beyond the scope of this book.

To help you create simple drawings, we have included a `graphics` module with the book that is a simplified version of Python's more complex library module. The module code and usage instructions are available with the source code for the book on its companion web site. In the following sections, you will learn all about this module, and how to use it for creating simple drawings that consist of basic geometric shapes and text.

You can make simple drawings out of lines, rectangles, and circles.