TRAINING & REFERENCE

murach's Python programming

2ND EDITION

(Chapter 2)

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • **www.murach.com**

What developers have said about previous editions

"This is by far the best Python tutorial I have come across. Tried Udemy, Udacity, some other video and printed tutorials, but didn't get the feel of it. Murach's side-by-side lecture & example really works for me."

Posted at an online bookseller

"This is not only a book to learn Python but also a guide to refer to whenever I face a Python problem. What I really like is that it is very well-organized and easy to follow, and it does not make you bored by providing unnecessary details."

Posted at an online bookseller

"The material in the book itself is worth 6 stars [out of 5]. It's second to none in its quality."

Posted at an online bookseller

"This is now my third text book for Python, and it is the ONLY one that has made me feel comfortable solving problems and reading code. The paired-pages approach is fantastic, and it makes learning the syntax, rules, and conventions understandable for me."

Posted at an online bookseller

"[Look at the contents] and see where they put testing/debug: Chapter 5. I love the placement! Folks going through the book get Python installed, receive positive returns with some basic code, and then take a breather with testing/debugging before more complex things come their way."

Jeremy Johnson, DreamInCode.net

"*Murach's Rocks*: Murach's usual excellent coverage with examples that are actually practical."

Posted at an online bookseller

2

How to write your first programs

The quickest and best way to *learn* Python programming is to *do* Python programming. That's why this chapter shows you how to write complete Python programs that get input, process it, and display output. When you finish this chapter, you'll have the skills for writing comparable programs of your own.

Basic coding skills

This chapter starts by presenting some basic coding skills. You'll use these skills in every program that you develop.

How to code statements

Figure 2-1 presents the rules for coding Python *statements*. To start, you should know that, unlike many programming languages, the indentation of each line matters in a Python program. With Python, the indentation is typically four spaces, as illustrated in the first example in this figure. As you learn how to code the various types of Python statements, you'll get the specifics for the indentation that's required.

In some cases, you'll want to divide a long statement over two or more lines. Then, you can use *implicit continuation*. To do that, you divide a statement before or after an operator like a plus or minus sign. You can also divide a statement after an opening parenthesis. When you divide the statement, it's a good practice to indent its continuation lines. This is illustrated by the implicit continuation example in this figure.

The other way to continue a statement is to use *explicit continuation*. Then, you code a backslash to show that a line is continued. In general, this is discouraged and isn't usually required, so you shouldn't need to use it often.

In this book, you'll see many examples of statements that require indentation. You'll also see many examples of implicit continuation. For now, you just need to have a general idea of what's required.

The first line in a Python program is often a *shebang line* like the one in this figure. This line is ignored by Windows, but Unix-like systems, including macOS, use this line to determine what interpreter to use to run the program. In this example, the shebang line tells the operating system to use Python 3. In this book, every program has been written for Python 3. As a result, they all use the shebang line shown in this figure.

If you're using IDLE to develop programs, you don't need to include the shebang line at all. However, it's generally considered a good practice to include this line since it clearly indicates what version of Python should be used to run the program. In addition, it can make it easier to run the program on Unix-like operating systems.

The Python code for a Test Scores program

```
#!/usr/bin/env python3

counter = 0
score_total = 0
test_score = 0

while test_score != 999:
    test_score = int(input("Enter test score: "))
    if test_score >= 0 and test_score <= 100:
        score_total += test_score
        counter += 1

average_score = round(score_total / counter)

print("Total Score: " + str(score_total))
print("Average Score: " + str(average_score))
```

An indentation error

```
print("Total Score: " + str(score_total))
 print("Average Score: " + str(average_score))
```

Two ways to continue one statement over two or more lines

Implicit continuation

```
print("Total Score: " + str(score_total)
    + "\nAverage Score: " + str(average_score))
```

Explicit continuation

```
print("Total Score: " + str(score_total) \
    + "\nAverage Score: " + str(average_score))
```

Coding rules

- Python relies on proper indentation. Incorrect indentation causes an error.
- The standard indentation is four spaces whenever it is required.
- With *implicit continuation*, you can divide statements after parentheses, brackets, and braces, and before or after operators like plus or minus signs.
- With *explicit continuation*, you can use the `\` character to divide statements anywhere on a line.

Description

- A *statement* performs a task. Each statement must be indented properly.
- A program typically starts with a *shebang line* that begins with a hash (`#`) symbol followed by a bang (`!`) symbol. This line identifies the interpreter to use when running the program.
- If you're using IDLE to run your programs, you don't need a shebang line. However, it's generally considered a good practice to include one.

Figure 2-1 How to code statements

How to code comments

To provide *comments* that describe portions of a program, you use the techniques in figure 2-2. To code a *block comment*, you code a hash symbol (#) at the start of each line. To code an *inline comment*, you code the hash symbol on the same line but after a Python statement. Since all comments are ignored by the Python interpreter, they have no effect on the operation of the program.

Most of the time, comments are used to document portions of code that are difficult to understand. This can be helpful to the programmer who develops the program as well as to programmers who maintain the program later.

Comments can also be used to disable statements that you don't want to run when you compile and run a program. This is called *commenting out* statements. To do that, you code a hash symbol before each statement that you want to comment out. Then, when you're ready to test those statements, you can *uncomment* them by removing the hash symbols.

When you're using IDLE, it's easy to comment out or uncomment statements. To do that, you can select the statements. Then, you can use the commands available from the Format menu to comment out or uncomment the selected statements.

The Test Scores program after comments have been added

```
#!/usr/bin/env python3

# This is a tutorial program that illustrates the use of the while
# and if statements

# initialize variables
counter = 0
score_total = 0
test_score = 0

# get scores
while test_score != 999:
    test_score = int(input("Enter test score: "))
    if test_score >= 0 and test_score <= 100:
        score_total += test_score          # add score to total
        counter += 1                       # add 1 to counter

# calculate average score
#average_score = score_total / counter
#average_score = round(average_score)
average_score = round(            # implicit continuation
    score_total / counter)        # same results as commented out statements

# display the result
print("======================")
print("Total Score: " + str(score_total)   # implicit continuation
      + "\nAverage Score: " + str(average_score))
```

Block comments

Inline comments

Commented out statements

Guidelines for using comments

- Use comments to describe portions of code that are hard to understand, but don't overdo them.
- Use comments to comment out (or disable) statements that you don't want to test.
- If you change the code that's described by comments, change the comments too.

Description

- *Comments* start with the **#** sign, and they are ignored by the compiler. *Block comments* are coded on their own lines. *Inline comments* are coded after statements to describe what they do.
- Comments are used to document what a program or portion of code does. This can be helpful not only to the programmer who creates the program, but also to those who maintain the program later on.
- Comments can also be used to *comment out* statements so they aren't executed when the program is tested. Later, the statements can be *uncommented* so the statements will be executed when the program is tested. This can be helpful when debugging.
- When you're using IDLE, you can use the Format menu to comment out and to uncomment the statements that you've selected.

Figure 2-2 How to code comments

How to use functions

A *function* is a group of statements that perform a specific task. Python provides many *built-in functions* that you'll use throughout this book. And you'll learn how to create your own functions in chapter 4.

To use one of the built-in functions, you use the syntax shown at the top of figure 2-3. In a syntax summary like this, the italicized words are ones that you have to supply and the brackets indicate portions of the code that are optional.

To *call* a function, then, you code the function name, a set of parentheses, and any *arguments* that are required by the function within the parentheses. If a function requires more than one argument, you separate the arguments with commas.

The print() function shows how this works. This function displays the data that is passed to it as arguments. But note that the arguments are optional. If no arguments are passed to this function, it prints a blank line.

This is illustrated by the calls to the print() function in the example. Here, the first print() function prints "Hello out there!" to the console. The second print() function prints a blank line (or skips a line). And the third print() function prints "Goodbye!".

The main point of this figure, though, is to show you how to call any function, not just the print() function. In this chapter, you'll learn more about using the print() function as well as five other functions. And the syntax for using these functions is always the same.

The syntax for calling any function

```
function_name([arguments])
```

The print() function

Function	Description
`print([data])`	Prints the data argument to the console followed by a new line character. If the call doesn't include a data argument, this function prints a blank line to the console.

A script with three statements

```
print("Hello out there!")
print()
print("Goodbye!")
```

The console after the program runs

```
Hello out there!

Goodbye!
```

Description

- A *function* is a reusable unit of code that performs a specific task.
- Python provides many *built-in functions* that do common tasks like getting input data from the user and printing output data to the console.
- When you *call a function*, you code the name of the function followed by a pair of parentheses. Within the parentheses, you code any *arguments* that the function requires, and you separate multiple arguments with commas.
- In a syntax summary like the one at the top of this page, brackets [ ] mark the portions of code that are optional. And the italicized portions of the summary are the ones that you have to supply.
- In this chapter, you'll learn to use the print() function plus five other functions.

Figure 2-3 How to use functions

How to work with data types and variables

When you develop a Python program, you work with variables that store data. In the topics that follow, you'll learn how to work with three of the most common Python data types.

How to assign values to variables

Figure 2-4 starts by summarizing three of the Python *data types*. The *str* (or *string*) data type holds any characters like "Mike" or "40". The *int* (or *integer*) data type holds whole numbers like 21 or -25. And the *float* (or *floating-point*) data type holds numbers with decimal places like 41.3 or -25.78.

When you work with these data types in a program, you normally assign them to *variables*. To do that, you code *assignment statements* like those in the first group of examples in this figure. These statements consist of a variable name like first_name or quantity1, an equals sign (the *assignment operator*), and the value that should be assigned to the variable. These variables and values are stored in the main memory of the computer.

In this figure, the first group of examples *initialize*, or assign initial values to, the variables. To do that, they assign *literal values* to the variables. Here, the *string literal* is coded within quotation marks, but the *numeric literals* aren't. That means that Mike is assigned to the first_name variable, 3 is assigned to the quantity1 variable, 5 is assigned to the quantity2 variable, and 19.99 is assigned to the list_price variable.

The second group of examples shows how to assign a new value to a variable. To start, the first statement sets the first_name variable to Joel. The second statement sets the value of quantity1 to 10. The third statement sets the value of the quantity1 variable to the value of the quantity2 variable, or 5. And the fourth statement sets the value of the quantity1 variable to a string value of "15".

The third example shows how to use *multiple assignment* to assign values to two or more variables using a single statement. To do that, you separate the assignments with commas. In most cases, a statement like this will be used to initialize related variables.

The last example shows that the names of variables are *case-sensitive*. That means an uppercase (or capital) letter is different than a lowercase letter. As a result, Quantity2 is different than quantity2. In this example, then, a NameError exception occurs because you can't set quantity1 equal to a variable that doesn't exist.

As you review this code, note that Python determines the data type for a variable based on the value that's assigned to the variable. For instance, in the second group of examples, the second statement assigns a numeric literal of 10 to the quantity1 variable. As a result, Python uses the int data type for this variable. However, the fourth statement assigns a string literal of "15" to the quantity1 variable. As a result, this variable is now assigned a value of the str type, not a value of the int type.

Three Python data types

Data type	Name	Examples				
`str`	String	`"Mike"`	`"40"`	`'Please enter name: '`		
`int`	Integer	`21`	`450`	`0`	`-25`	
`float`	Floating-point	`21.9`	`450.25`	`0.01`	`-25.2`	`3.1416`

Code that initializes variables

```
first_name = "Mike"    # sets first_name to a str value of "Mike"
quantity1 = 3          # sets quantity1 to an int value of 3
quantity2 = 5          # sets quantity2 to an int value of 5
list_price = 19.99     # sets list_price to a float value of 19.99
```

Code that assigns new values to the variables above

```
first_name = "Joel"    # sets first_name to a str of "Joel"
quantity1 = 10         # sets quantity1 to an int of 10
quantity1 = quantity2  # sets quantity1 to an int of 5
quantity1 = "15"       # sets quantity1 to a str of "15", not an int of 15
```

Code that assigns values to two variables

```
quantity1 = 3, list_price = 19.99
```

Code that causes an error because of incorrect case

```
quantity1 = Quantity2  # NameError: 'Quantity2' is not defined
```

How to code literal values

- To code a *literal value* for a string, enclose the characters of the string in single or double quotation marks. This is called a *string literal*.
- To code a literal value for a number, code the number without quotation marks. This is called a *numeric literal*.

Description

- A *variable* can change, or *vary*, as code executes.
- A *data type* defines the type of data for a value.
- An *assignment statement* uses the equals sign (`=`) to assign a value to a variable. The value can be a literal value, another variable, or an expression like the arithmetic expressions in the next figure.
- To *initialize* a variable, you assign an initial value to it. Then, you can assign different values later in the program.
- You can assign a value of any data type to a variable, even if that variable has previously been assigned a value of a different data type.
- Because variable names are *case-sensitive*, you must be sure to use the correct case when coding the names of variables.

Figure 2-4 How to assign values to variables

How to name variables

Figure 2-5 presents the Python rules for creating variable names. It also lists the Python *keywords* that should *not* be used for variable names. That's because these words are used in other ways by Python.

Besides these rules, though, you should give your variables meaningful names. That means that it should be easy to tell what a variable name refers to and easy to remember how to spell the name. To create names like that, you should avoid abbreviations. If, for example, you abbreviate the name for monthly investment as mon_inv, it's hard to tell what it refers to and hard to remember how you spelled it. But if you spell it out as monthly_investment, both problems are solved.

Similarly, you should avoid abbreviations that are specific to one industry or field of study unless you are sure the abbreviation is widely understood. For example, mpg is a common abbreviation for miles per gallon, but cpm could stand for a number of things and should be spelled out.

To create an identifier that has more than one word in it, most Python programmers use underscores to separate the words in a variable name. For example, the variables named tax_rate and monthly_investment use underscores to separate words. This is known as *underscore notation,* or *snake case*.

However, some programmers use a convention called *camel case*. With this convention, the first letter of each word is uppercase except for the first word. For example, the variables named taxRate and monthlyInvestment use camel casing.

In this book, sections 1 and 2 use underscore notation for most variable names. However, since camel case is commonly used with object-oriented programs, sections 3 and 4 use camel casing for all of the object-oriented modules. When you're writing your own programs, you can choose the convention that you prefer. What's most important is to pick one of these conventions and use it consistently within a module.

Rules for naming variables

- A variable name must begin with a letter or underscore.
- A variable name can't contain spaces, punctuation, or special characters other than the underscore.
- A variable name can't begin with a number, but can use numbers later in the name.
- A variable name can't be the same as a *keyword* that's reserved by Python.

Python keywords

```
and         except     lambda      while
as          False      None        with
assert      finally    nonlocal    yield
break       for        not
class       from       or
continue    global     pass
def         if         raise
del         import     return
elif        in         True
else        is         try
```

Two naming styles for variables

```
variable_name          # underscore notation
variableName           # camel case
```

Recommendations for naming variables

- Start all variable names with a lowercase letter.
- Use underscore notation or camel case.
- Use meaningful names that are easy to remember.
- Don't use the names of built-in functions, such as print().

Description

- When naming variables, you must follow the rules shown above. Otherwise, you'll get errors when you try to run your code.
- It's also a best practice to follow the naming recommendations shown above. This makes your code easier to read, maintain, and debug.
- With *underscore notation*, all letters are lowercase with words separated by underscores. This can also be referred to as *snake case*.
- With *camel case*, the first word is lowercase and subsequent words start with a capital letter.

Figure 2-5 Rules and recommendations for naming variables

How to work with numeric data

The next three figures show you how to work with numeric data. That's something you will do in just about every program that you write.

How to code arithmetic expressions

Figure 2-6 shows how to code *arithmetic expressions*. These expressions consist of two or more *operands* that are operated upon by *arithmetic operators*. The operands in an expression can be either numeric variables or numeric literals.

The first table in this figure summarizes the arithmetic operators. Here, the `+`, `-`, and `/` operators are the same as those used in basic arithmetic, and the `*` operation is used for multiplication.

Python also has an operator for integer division (`//`) that truncates the decimal portion of the division. It has a *modulo operator* (`%`), or *remainder operator*, that returns the remainder of a division. And it has an *exponentiation operator* (`**`) that raises a number to the specified power.

The second table in this figure shows some examples of how these operators work. For the most part, this is simple arithmetic. Here, the division operator always returns a floating point number. However, the integer division operator truncates the result and returns an integer, and the modulo operator returns just the remainder of an integer division operation. As a result, 25 divided by 4 is 6.25, but 25 divided by 4 with integer division is 6. And 25 modulo 4 is 1 because that's the remainder.

When an expression includes two or more operators, the *order of precedence* determines which operators are applied first. This order is summarized in the table in this figure. For instance, Python performs all multiplication and division operations from left to right before it performs any addition and subtraction operations.

If you need to override the default order of precedence, you can use parentheses. Then, Python performs the expressions in the innermost sets of parentheses first, followed by the expressions in the next sets of parentheses, and so on. This works the same as it does in algebra, and this is typical of all programming languages. The examples in the last table show how this works.

Python's arithmetic operators

Operator	Name	Description
`+`	Addition	Adds two operands.
`-`	Subtraction	Subtracts the right operand from the left operand.
`*`	Multiplication	Multiplies two operands.
`/`	Division	Divides the right operand into the left operand. The result is always a floating-point number.
`//`	Integer division	Divides the right operand into the left operand and drops the decimal portion of the result.
`%`	Modulo / Remainder	Divides the right operand into the left operand and returns the remainder. The result is always an integer.
`**`	Exponentiation	Raises the left operand to the power of the right operand.

Examples with two operands

Example	Result
`5 + 4`	`9`
`25 / 4`	`6.25`
`25 // 4`	`6`
`25 % 4`	`1`
`3 ** 2`	`9`

The order of precedence for arithmetic expressions

Order	Operators	Direction
1	`**`	Left to right
2	`* / // %`	Left to right
3	`+ -`	Left to right

Examples that show the order of precedence and use of parentheses

Example	Result
`3 + 4 * 5`	`23` (the multiplication is done first)
`(3 + 4) * 5`	`35` (the addition is done first)

Description

- An *arithmetic expression* consists of one or more *operands* that are operated upon by *arithmetic operators*.
- You don't have to code spaces before and after the arithmetic operators.
- When an expression mixes integer and floating-point numbers, Python converts the integers to floating-point numbers.
- If you use multiple operators in one expression, you can use parentheses to clarify the sequence of operations. Otherwise, Python applies its *order of precedence*.

Figure 2-6 How to code arithmetic expressions

How to use arithmetic expressions in assignment statements

Now that you know how to code arithmetic expressions, figure 2-7 shows how to use these expressions with variables and assignment statements. Here, the first two examples show how you can use the multiplication and addition operators in Python statements.

This is followed by a table that presents three of the *compound assignment operators*. These operators provide a shorthand way to code common assignment statements. For instance, the `+=` operator modifies the value of the variable on the left of the operator by adding the value of the expression on the right to the value of the variable on the left. When you use this operator, the variable on the left must already have been initialized.

The other two operators in this table work similarly, but the `-=` operator subtracts the result of the expression on the right from the variable on the left. And the `*=` operator multiples the variable on the left by the result of the expression on the right.

The first example after this table shows two ways to increment a variable by adding 1 to it, a common coding requirement. Here, the first statement assigns a value of 0 to a variable named counter. Then, the second statement uses an arithmetic expression to add 1 to the value of the counter. This shows how you can code a variable name on both sides of the `=` operator. By contrast, the third statement adds 1 to the counter by using the `+=` operator. When you use the `+=` operator, you don't need to code the variable name on the right side of the `=` operator, which makes the code more concise.

The second example after the table shows how the `+=` operator can be used to add variable values to a variable that stores the total of the values. This is a common programming technique that is used in many of the programs in this book. It's followed by an example that shows how all three compound assignment operators work.

The last example illustrates a problem with float values that you should be aware of. Because float values are stored internally as floating-point numbers, the results of arithmetic operations aren't exact. In this example, the tax result is 7.495000000000001, even though it should be 7.495. For now, you just need to be aware of the potential for inaccuracies like this so you won't be surprised by them. Later, you'll learn ways to deal with these inaccuracies. One way to do that is to round the results of calculations as shown later in this chapter.

Code that calculates sales tax

```
subtotal = 200.00
tax_percent = .05
tax_amount = subtotal * tax_percent        # 10.0
grand_total = subtotal + tax_amount        # 210.0
```

Code that calculates the perimeter of a rectangle

```
width = 4.25
length = 8.5
perimeter = (2 * width) + (2 * length)     # 8.5 + 17 = 25.5
```

The most useful compound assignment operators

Operator	Description
`+=`	Adds the result of the expression to the variable.
`-=`	Subtracts the result of the expression from the variable.
`*=`	Multiplies the variable value by the result of the expression.

Two ways to increment the number in a variable

```
counter = 0
counter = counter + 1           # counter = 1
counter += 1                    # counter = 2
```

Code that adds two numbers to a variable

```
score_total = 0                 # score_total = 0
score_total += 70               # score_total = 70
score_total += 80               # score_total = 150
```

More statements that use the compound assignment operators

```
total = 1000.0
total += 100.0                  # total = 1100.0

counter = 10
counter -= 1                    # counter = 9

price = 100
price *= .8                     # price = 80.0
```

A floating-point result that isn't precise

```
subtotal = 74.95                # subtotal = 74.95
tax = subtotal * .1             # tax = 7.495000000000001
```

Description

- Besides the assignment operator (`=`), Python provides for *compound assignment operators*. These operators are a shorthand way to code common assignment operations.
- Besides the compound operators in the table, Python offers `/=`, `//=`, `%=`, and `**=`.
- When working with floating-point numbers, be aware that they are approximations, not exact values. This can cause inaccurate results.

Figure 2-7 How to use arithmetic expressions in assignment statements

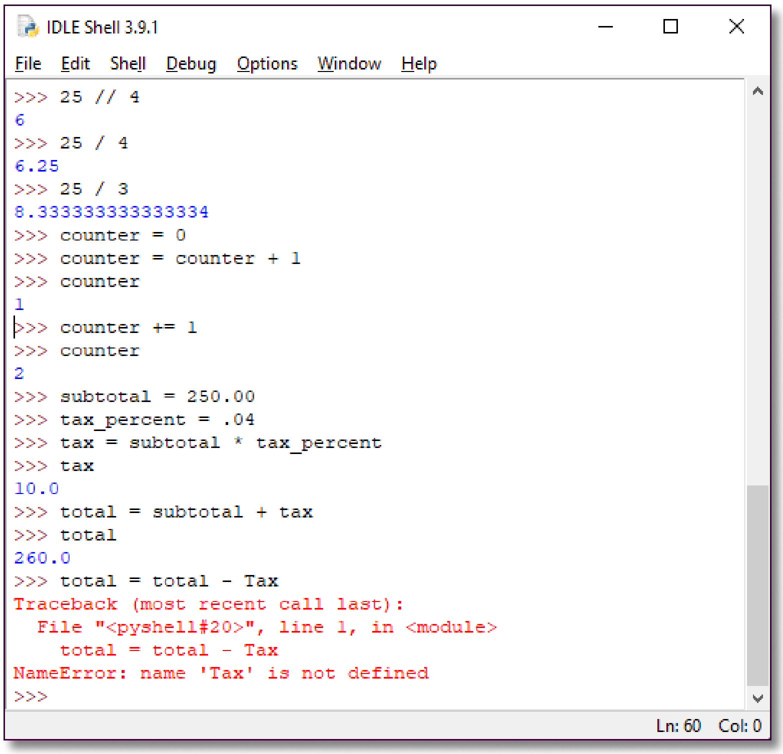
How to use the interactive shell for testing numeric operations

Remember the IDLE interactive shell that you learned about in the previous chapter? If you're confused by any of the examples presented in the previous figures, now is a good time to use it! To do that, open the interactive shell as described in the previous chapter. Then, start experimenting with the arithmetic operations as shown in figure 2-8.

In fact, you probably should take a break right now and do some experimenting, especially if you have any doubts about how the arithmetic operations work. For instance, test the use of each arithmetic operator. Test the use of each compound assignment operator. And test the results of floating-point operations. Along the way, you'll probably get some error messages, but that's all part of the learning process. Each one should give you a better understanding of how Python works.

When you get an error, you might want to attempt to fix the entry that caused the error. To do that, you can start by pressing Alt+p (Windows) or Command+p (macOS) to retype the previous entry at the prompt. Then, you can attempt to fix the problem by editing your previous entry and pressing the Enter key to execute it again.

The Python interactive shell as it's used for numeric testing



```
>>> 25 // 4
6
>>> 25 / 4
6.25
>>> 25 / 3
8.333333333333334
>>> counter = 0
>>> counter = counter + 1
>>> counter
1
>>> counter += 1
>>> counter
2
>>> subtotal = 250.00
>>> tax_percent = .04
>>> tax = subtotal * tax_percent
>>> tax
10.0
>>> total = subtotal + tax
>>> total
260.0
>>> total = total - Tax
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    total = total - Tax
NameError: name 'Tax' is not defined
>>>
```

How to use the shell

- To test a statement, type it at the prompt and press the Enter key. You can also type the name of a variable at the prompt to see what its value is.
- Any variables that you create remain active for the current session. As a result, you can use them in statements that you enter later in the same session.
- To retype your previous entry, press Alt+p (Windows) or Command+p (macOS).
- To cycle through all of the previous entries, continue pressing the Alt+p (Windows) or Command+p (macOS) keystroke until the entry you want is displayed at the prompt.

Description

- The Python interactive shell that you learned about in chapter 1 is an excellent tool for learning more about the way numeric operations work. For example, you can use it to test assigning numeric literals and arithmetic expressions to variables.
- To see how numeric statements work, just enter a statement or series of statements and see what the results are.
- This type of testing will give you a better idea of what's involved as you work with integer and floating-point numbers. It will also show your errors when you enter a statement incorrectly so you will learn the proper syntax as you experiment.

Figure 2-8 How to use the interactive shell for testing numeric operations

How to work with string data

The next three figures show you how to work with string data. That's another thing that you will do in just about every program you write.

How to assign strings to variables

In figure 2-4, you learned how to assign strings to variables. Now, figure 2-9 expands upon that skill. Here, the first group of examples shows that you can use single or double quotes to create a string literal. You can also create an *empty string* by coding a set of quotation marks with nothing between them. And you can assign a new value to a string variable. This works the same way it does with numeric variables.

How to join strings

This figure also shows how to *concatenate*, or *join*, strings. To do that, you can use either the **+** operator or an *f-string* as shown in the second set of examples in this figure. To join strings using the **+** operator, you code the names of the variables and string literals you want to join. Here, the last_name variable is joined with a string literal that contains a comma and a space, and then that's joined with the first_name variable.

To join strings using an f-string, you code a string literal preceded by the letter *f*. Within the string literal, you code variables within braces. Then, everything outside the braces is treated as a literal. In this example, the last_name and first_name variables are coded in braces, and the comma and space are treated as a literal. The result is the same as when you use the **+** operator, but the code is more concise.

F-strings are particularly useful when you want to join a string and a number. This is illustrated by the next set of examples. To join a string and a number with the **+** operator, you need to use the str() function to convert the number to a string before you can join it. Here, the name variable is joined with " is ", the age variable, and " years old." This code uses the str() function to explicitly convert the age variable to a string before the join takes place. The resulting string is:

```
Bob Smith is 40 years old.
```

Note that if you don't code the str() function, a TypeError exception occurs with a message that says Python can't implicitly convert an int object to a string.

When you join a string with a number using an f-string, it isn't necessary to use the str() function. Instead, the conversion is done implicitly. As you can see, that simplifies the code for the join operation.

If you use joins to create long strings, you will sometimes want to code the statement over two or more lines. To do that, you can use implicit continuation as shown in the last set of examples in this figure. To do that when you use the **+** operator, you can divide the statement before or after the plus signs. To do that when you use f-strings, you code an f-string for the string literal on each line.

The str() function for converting numbers to strings

Function	Description
`str(data)`	Converts a numeric argument to a string and returns the string.

How to assign strings to variables

```
first_name = "Bob"                    # first_name = Bob
last_name = 'Smith'                   # last_name = Smith
name = ""                             # name = empty string
name = "Bob Smith"                    # name = Bob Smith
```

How to join strings

With the + operator

```
name = last_name + ", " + first_name    # name is "Smith, Bob"
```

With an f-string

```
name = f"{last_name}, {first_name}"     # name is "Smith, Bob"
```

How to join a string and a number

```
name = "Bob Smith"
age = 40
```

With the + operator and the str() function

```
message = name + " is " + str(age) + " years old."
```

With an f-string

```
message = f"{name} is {age} years old."    # str() function not needed
```

Implicit continuation of a string over two or more coding lines

With the + operator

```
print("Name:  " + name + "\n" +
      "Age:   " + str(age))
```

With an f-string

```
print(f"Name:  {name}\n"
      f"Age:   {age}")
```

Description

- A *string* can consist of one or more characters, including letters, numbers, and special characters like `*`, `&`, and `#`.
- To specify the value of a string, you can enclose the text in either double or single quotation marks. This is known as a *string literal*.
- To assign an *empty string* to a variable, you code a set of quotation marks with nothing between them. This means that the string doesn't contain any characters.
- To join strings, you can use the `+` operator or an *f-string*. To use an f-string, you code an *f* before a string literal. Then, you use braces (`{}`) to identify the variables to insert into the string literal.

Figure 2-9 How to assign strings to variables and how to join strings

How to include special characters in strings

Figure 2-10 shows how to use *escape sequences* to include special characters in a string. To start, the table shows six of the many escape sequences that you can use with Python. These sequences let you put characters in a string that you can't put in just by pressing the appropriate key on the keyboard. For instance, the \n escape sequence is similar to pressing the Enter key in the middle of a string.

Escape sequences are needed so the Python compiler can interpret the code correctly. For instance, since single and double quotations marks are used to identify strings, coding them within the strings can cause syntax errors. But when the quotation marks are preceded by escape characters, the Python compiler can interpret them correctly.

The first three examples in this figure show how escape characters work in print() functions. In the first two examples, the new line sequence is used to start a new line. In the second example, the tab sequence is also used. And in the third example, the backslash sequence is used.

The last example shows the four ways that you can put quotation marks in a string, even though quotation marks are used to identify a string. The first two statements use escape sequences to insert the marks in a string. The third statement uses single quotes within a string that's marked by double quotes. And the last statement does the reverse.

Common escape sequences

Sequence	Character
`\n`	New line
`\t`	Tab
`\r`	Return
`\"`	Quotation mark in a double quoted string
`\'`	Quotation mark in a single quoted string
`\\`	Backslash

The new line character

```
print("Title: Python Programming\nQuantity: 5")
```

Displayed on the console

```
Title: Python Programming
Quantity: 5
```

The tab and new line characters

```
print("Title:\t\tPython Programming\nQuantity:\t5")
```

Displayed on the console

```
Title:          Python Programming
Quantity:       5
```

The backslash in a Windows path

```
print("C:\\murach\\python")
```

Displayed on the console

```
C:\murach\python
```

Four ways to include quotation marks in a string

```
"Type \"x\" to exit"   # String is: Type "x" to exit.
'Type \'x\' to exit'   # String is: Type 'x' to exit.
"Type 'x' to exit"     # String is: Type 'x' to exit.
'Type "x" to exit'     # String is: Type "x" to exit.
```

Description

- Within a string, you can use *escape sequences* to include certain types of special characters such as new lines and tabs. You can also use escape characters to include special characters such as quotation marks and backslashes.
- Another way to include quotation marks in a string is to code single quotes within a string that's in double quotes, or vice versa.

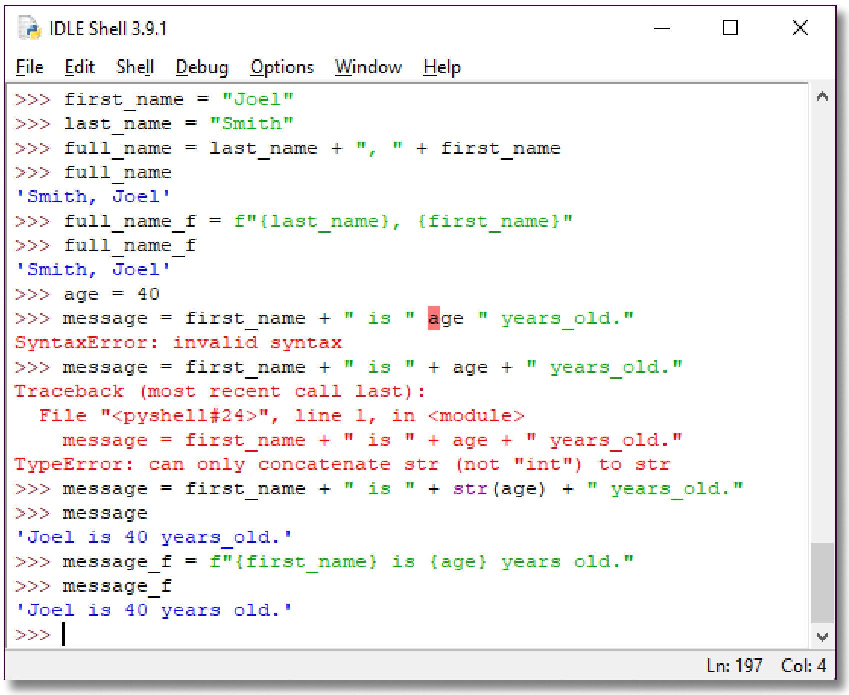
Figure 2-10 How to include special characters in strings

How to use the interactive shell for testing string operations

If you have any doubts about how the string operations work, this is a good time to take a break and do some experimenting with them. Figure 2-11 shows how.

As you experiment, you'll probably get a few error messages, but each one should give you a better understanding of how Python works. For instance, can you tell what's wrong with the statement that caused the SyntaxError in this figure? And can you tell what's wrong with the statement that caused the TypeError exception? If there's any doubt, you can retype the statement that caused the error, modify it, and press Enter to execute it again. The more you experiment, the more you'll learn.

The Python interactive shell as it's used for string testing



```
>>> first_name = "Joel"
>>> last_name = "Smith"
>>> full_name = last_name + ", " + first_name
>>> full_name
'Smith, Joel'
>>> full_name_f = f"{last_name}, {first_name}"
>>> full_name_f
'Smith, Joel'
>>> age = 40
>>> message = first_name + " is " age " years_old."
SyntaxError: invalid syntax
>>> message = first_name + " is " + age + " years_old."
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    message = first_name + " is " + age + " years_old."
TypeError: can only concatenate str (not "int") to str
>>> message = first_name + " is " + str(age) + " years_old."
>>> message
'Joel is 40 years_old.'
>>> message_f = f"{first_name} is {age} years old."
>>> message_f
'Joel is 40 years old.'
>>> 
```

How to use the shell

- You can use the same skills to test string operations in the interactive shell as you do to test numeric operations. See figure 2-8 for more information.

Description

- The Python interactive shell that you learned about in chapter 1 is an excellent tool for learning more about the way string operations work. For example, you can use it to test assigning strings to variables, joining strings, and including special characters in strings.
- To see how string statements work, just enter a statement or series of statements and see what the results are. This will also show your errors when you enter a statement incorrectly so you will learn the proper syntax as you experiment.

Figure 2-11   How to use the interactive shell for testing string operations

How to use five of the Python functions

When you program with Python, you will use many of its built-in functions. So far, you've been introduced to the print() and str() functions. Now, you'll learn more about the print() function, and you'll learn how to use four other functions.

How to use the print() function

Figure 2-12 starts by showing the syntax for the print() function. In most cases, you'll just code one or more data items as the arguments when you use this function. Although you can also code sep and end arguments, you won't need them in your early programs so don't worry about them right now.

The first group of examples shows print() functions that receive one or more arguments. In this case, the numbers are printed directly to the console. For instance, the first function prints a number. The second function prints a string and a number and automatically puts a space between the two values. And the third function prints four integers with spaces between them.

The next group of examples shows two different ways to get the same results when using print() functions. The first print() function receives four arguments and works like the previous examples. It prints the four values with spaces between them. Note, however, that the escape sequence for a new line is coded at the start of the third argument so that argument is displayed on a new line.

The second example gets the same result by passing a single string argument to the print() function. This string consists of a string, a number, another string, and another number that have been joined together with an f-string.

When you start using print() functions, you will probably want to use the first method of coding because it's easier. But as you get more experience, you'll find that the second method isn't much harder, and it lets you get results that you can't get with the first method.

The last group of examples in this figure shows how to use the sep and end arguments. Since you must code the names of these arguments, they're called *named arguments*. Here, the first statement uses the sep argument to separate the integers with a bar character that has a space before and after it. And the second statement uses the end argument to put three exclamation points at the end of the data, instead of ending the data with a new line character. If necessary, you can code both the sep and end arguments in the same function call.

The syntax of the print() function

```
print(data[, sep=' '][, end='\n'])
```

Three print() functions that receive one or more arguments

```
print(19.99)                             # 19.99
print("Price:", 19.99)                   # Price: 19.99
print(1, 2, 3, 4)                        # 1 2 3 4
```

Two ways to get the same result

A print() function that receives four arguments

```
print("Total Score:", score_total,
      "\nAverage Score:", average_score)
```

A print() function that receives one string as the argument

```
print(f"Total Score: {score_total}"
      f"\nAverage Score: {average_score}")
```

The data that's displayed by both functions

```
Total Score: 240
Average Score: 80
```

Examples that use the sep and end arguments

```
print(1,2,3,4,sep=' | ')                 # 1 | 2 | 3 | 4
print(1,2,3,4,end='!!!')                 # 1 2 3 4!!!
```

Description

- The print() function can accept one or more data arguments.
- If you pass a series of arguments to the print() function, numbers don't have to be converted to strings.
- If you pass just one string as the argument for a print() function, numbers have to be converted to strings within the string argument.
- In a syntax summary, an argument that begins with a name and an equals sign (`=`) is a *named argument*. To pass a named argument to a function, you code the name of the argument, an equals sign (`=`), and the value of the argument.
- The print() function provides a sep argument that can change the character that's used for separating the strings from one space to something else.
- The print() function also provides an end argument that can change the ending character for a print() function from a new line character (`\n`) to something else.

Figure 2-12 How to use the print() function

How to use the input() function

Most programs get input data from the user and then adjust the processing based on that input. For a program that uses the console, you can use the input() function to get that data. Figure 2-13 shows how.

The input() function can take one string argument that prompts the user to enter data. In the first example, this argument is “Enter your first name:”. Then, when Python executes the function, it displays this *prompt* on the console, and the program pauses as it waits for the user to make an entry.

When the user makes that entry and presses the Enter key, the entry is returned by the function so it can be saved in a variable. In this example, it is saved in a variable named first_name.

The second example shows how you can get an entry without using the prompt argument. In this case, you use a print() function to display the prompt. Then, you use an input() function to get the data for that prompt. When you get data this way, the prompt is displayed on one line and the entry on the next.

When you use the input() function, remember that all entries are returned as strings. This is illustrated by the last example. If the user enters 85, and that number is stored in a variable, it is stored with the str data type. Then, if the program tries to add that variable to a variable with a numeric data type, a TypeError exception occurs because you can’t add a string to a number.

The input() function for getting data from the console

Function	Description
`input([prompt])`	Pauses the program and waits for the user to enter data at the console. When the user presses Enter, this function returns the data entered by the user as a str value. If the call includes a prompt argument, this function prints the prompt to the console before pausing to wait for the user to enter data.

Code that gets string input from the user

```
first_name = input("Enter your first name: ")
print(f"Hello, {first_name}!")
```

The console

```
Enter your first name: Mike
Hello, Mike!
```

Another way to get input from the user

```
print("What is your first name?")
first_name = input()
print(f"Hello, {first_name}!")
```

The console

```
What is your first name?
Mike
Hello, Mike!
```

Code that attempts to get numeric input from the user

```
score_total = 0
score = input("Enter your score: ")
score_total += score      # causes an error because score is a string
```

Description

- You can use the input() function to get user input from the console. Typically, you assign the string that's returned by this function to a variable.
- The input() function always returns string data, even if the user enters a number.

Figure 2-13 How to use the input() function

How to use the int(), float(), and round() functions

Figure 2-14 presents three more functions. The int() and float() functions convert the data argument, which is typically a str value, to an int or float value. This is illustrated by the first three groups of examples. In the second and third examples, the input() function returns strings for numeric entries. Then, the strings are converted to int and float values.

By contrast, the round() function rounds a numeric value to the specified number of digits. This is illustrated by the last example. Here, a floating-point number is rounded to two decimal places. By rounding a float value, you can avoid the numeric inaccuracies that can occur when you're working with floating-point data.

How to chain functions

The second part of the second and third examples in this figure shows how you can *chain functions*. To do that, you code one function as the argument of another function. In the chained part of the second example, the input() function is coded as the argument of the int() function. As a result, two functions are chained together in a single statement. In the chained part of the third example, the input() function is coded as the argument of the float() function.

In the fourth example, an arithmetic expression is coded as the first argument of the round() function. Since the second argument of the round() function is 2, the result is rounded to two decimal places. This isn't chaining, but it shows how an arithmetic expression can be coded as the argument of a function.

At first, chaining may seem to make the code more complicated. But chaining is commonly used by professional programmers, so you need to get used to it. A good way to do that is to start by coding your programs without chaining. Then, when you have that working, you can use chaining to combine statements.

Three functions for working with numbers

Function	Description
`int(data)`	Converts the data argument to the int type and returns the int value.
`float(data)`	Converts the data argument to the float type and returns the float value.
`round(number [,digits])`	Rounds the number argument to the number of decimal digits in the digits argument. If no digits are specified, it rounds the number to the nearest integer.

Code that causes an exception

```
x = 15
y = "5"
z = x + y # TypeError: can't add an int to a str
```

How using the int() function fixes the exception

```
x = 15
y = "5"
z = x + int(y)                          # z is 20
```

Code that gets an int value from the user

```
quantity = input("Enter the quantity: ")   # quantity is str type
quantity = int(quantity)                   # quantity is int type
```

How to use chaining to get the int value in one statement

```
quantity = int(input("Enter the quantity: "))
```

Code that gets a float value from the user

```
price = input("Enter the price: ")         # price is str type
price = float(price)                       # price is float type
```

How to use chaining to get the float value in one statement

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

Code that uses the round() function

```
miles_driven = 150
gallons_used = 5.875
mpg = miles_driven / gallons_used          # mpg = 25.53191489361702
mpg = round(mpg, 2)                        # mpg = 25.53
```

How to combine the last two statements

```
mpg = round(miles_driven / gallons_used, 2)
```

Description

- If you try to perform an arithmetic operation on a string, an exception occurs. To fix that, you can use the int() and float() functions.
- When you *chain functions*, you code one function as the argument of another function. This is a common coding practice.

Figure 2-14 How to use the int(), float(), and round() functions

Two illustrative programs

This chapter ends by presenting two programs that illustrate the skills that have been presented. This is a good checkpoint for you. If you understand all of the code in these programs, you're on your way.

The Miles Per Gallon program

Figure 2-15 presents a Miles Per Gallon program. In the console, the user enters two values: the number of miles that were driven and the gallons of gas that were used. Then, the program calculates and displays the miles per gallon. Incidentally, in this console and in all of the console examples in this book, the user entries aren't boldfaced, but the characters displayed by the program are boldfaced.

In the Python code for this program, the first line is a shebang line that's appropriate for programs that are written for Python 3. This is followed by two print() functions that print a title and a blank line to the console.

Next, the program gets two input values from the user and stores them in two variables. In the input() functions, the prompts use escape sequences for tab characters. In addition, each input() function is chained within a float() function. As a result, both of these variables store float values, not str values.

Then, the program calculates the miles per gallon by dividing miles driven by gallons used. This is followed by a statement that rounds the miles per gallon value to two decimal places.

Last, the program uses four print() functions to display the miles per gallon value and a message that indicates that the program is ending. Here, the second print() function uses escape sequences for tab characters. In addition, it uses an f-string to convert the float value for miles per gallon to a string.

When you test a program like this, you should know that you need to enter valid numeric data whenever it is called for. Otherwise, the float() function won't be able to convert the strings to float values. That will cause a ValueError exception, and the program will crash.

For now, you have to accept this weakness of the programs that you write. In chapter 8, though, you'll learn how to handle exceptions and prevent this type of error.

The console

```
The Miles Per Gallon program

Enter miles driven:             150
Enter gallons of gas used:      35.875

Miles Per Gallon:               4.18

Bye!
```

The code

```
#!/usr/bin/env python3

# display a title
print("The Miles Per Gallon program")
print()

# get input from the user
miles_driven = float(input("Enter miles driven:\t\t"))
gallons_used = float(input("Enter gallons of gas used:\t"))

# calculate and round miles per gallon
mpg = miles_driven / gallons_used
mpg = round(mpg, 2)

# display the result
print()
print(f"Miles Per Gallon:\t\t{mpg}")
print()
print("Bye!")
```

Description

- This program uses many of the features presented in this chapter.
- The input() function is chained with the float() function to convert the user entries from str values to float values.
- This round() function rounds the result of the calculation to 2 decimal places.
- If the user enters a non-numeric value such as “sixty”, this program will crash and display an exception to the console. That’s because the input entry can’t be converted to a float value. You’ll learn how to fix an exception like that in chapter 8.

Figure 2-15 The Miles Per Gallon program

The Test Scores program

Figure 2-16 presents another program that illustrates the skills that are presented in this chapter. This time, the user enters three test scores. Then, the program calculates the average score and displays both the total of the scores and the average score.

You should be able to understand this program without any further explanation. So take some time to study it and see whether there's anything you don't understand. If so, here's a description of what's going on.

The four print statements at the start display the first four lines shown in the console. Then, a total_score variable is set to zero, and three input() functions are used to get three score entries from the user. These entries are converted to integers by the chained int() function, and they are added to the total_score variable by using the **+=** operator.

Next, the average score is calculated by dividing the total score by 3. This arithmetic expression is coded as the argument of a round() function that rounds to zero decimal places. Last, the results are printed, followed by a blank line and "Bye!".

This time, the print() function that displays the results receives four arguments, not an f-string. You can see how this print() function formats the data by reviewing the output in the console. You can also see how implicit continuation is used by dividing the statement after a comma.

Like the previous program, this one will crash if the user enters data that can't be converted to integers. So be sure to enter valid integer data when you test this program.

The console

```
The Test Scores program

Enter 3 test scores
======================
Enter test score: 75
Enter test score: 85
Enter test score: 95
======================
Total Score:   255
Average Score: 85

Bye!
```

The code

```
#!/usr/bin/env python3

# display a title
print("The Test Scores program")
print()
print("Enter 3 test scores")
print("======================")

# get scores from the user and accumulate the total
total_score = 0       # initialize the variable for accumulating scores
total_score += int(input("Enter test score: "))
total_score += int(input("Enter test score: "))
total_score += int(input("Enter test score: "))

# calculate average score
average_score = round(total_score / 3)

# format and display the result
print("======================")
print("Total Score:  ", total_score,
      "\nAverage Score:", average_score)
print()
print("Bye!")
```

Description

- This program shows how you can use a variable named total_score to accumulate the total for a series of entries. In this case, the program doesn't store the individual scores in variables. Instead, it just adds them to the total_score variable.
- If the user doesn't enter a valid integer for each test score, this program will crash and display an exception on the console. You'll learn how to fix that in chapter 8.

Figure 2-16 The Test Scores program

Perspective

The goal of this chapter has been to get you started with Python programming and to get you started fast. Now, if you understand the Miles Per Gallon and Test Scores programs, you've come a long way. You should also be able to write comparable programs of your own.

Keep in mind, though, that this chapter is just an introduction to Python programming. So in the next chapter, you'll learn how to code the control statements that drive the logic of your programs. This will take your programs to another level.

Terms

statement
implied continuation
explicit continuation
shebang line
comment
block comment
inline comment
comment out
uncomment
built-in function
function
call a function
argument
data type
string
str data type
integer
int data type
floating-point number
float data type
variable
assignment statement
assignment operator
initialize a variable
literal
string literal
numeric literal
multiple assignment
case-sensitive
keyword
underscore notation
snake case
camel case
arithmetic expression
operand
arithmetic operator
modulo operator
remainder operator
exponentiation operator
order of precedence
compound assignment operator
empty string
concatenate
join
f-string
escape sequence
named argument
prompt
chain functions

Summary

- A Python *statement* has a simple syntax that relies on indentation. A statement can be continued by using *implicit continuation*.
- Python *comments* can be *block* or *inline*. Comments can also be used to *comment out* portions of code so they won't be executed. Later, the code can be *uncommented* and executed.
- Python provides many *built-in functions* that you can *call* in your programs. When you call a function, you can include any *arguments* that are used by the function.
- Python *variables* are used to store data that changes as a program runs, and you use *assignment statements* to assign values to variables. Variable names are case-sensitive, and they are usually coded with *underscore notation*, also called *snake case,* or *camel case*.
- You can use *multiple assignment* to assign values to more than one variable using a single statement.
- The str, int, and float *data types* store values for strings, integers, and floating-point numbers.
- When you assign a value to a numeric variable, you can use *arithmetic expressions* that include *arithmetic operators*, variable names, and *numeric literals*.
- When you assign a value to a string variable, you can use the **+** operator or an *f-string* to join variables and *string literals*. Within a string literal, you can use *escape sequences* to provide special characters.
- The print() function prints output to the console, and the input() function gets input from the console. The str(), int(), and float() functions convert data from one data type to another. And the round() function rounds a number to the specified number of decimal places.
- To *chain functions*, you code one function as the argument of another function. This is a common coding practice.

Before you do the exercises for this book...

Before you do any of the exercises in this book, you need to install Python. In addition, you need to install the source code for this book from our website (www.murach.com). For details, see the appendix for your operating system.

Exercise 2-1 Modify the Miles Per Gallon program

In this exercise, you'll test and modify the code for the Miles Per Gallon program in figure 2-15. When you're finished, the program will get another user entry and do two more calculations, so the console will look something like this:

```
Enter miles driven:              150
Enter gallons of gas used:       15
Enter cost per gallon:           3

Miles Per Gallon:                10.0
Total Gas Cost:                  45.0
Cost Per Mile:                   0.3
```

If you have any problems when you test your changes, please refer to figure 1-9 of the last chapter, which shows how to fix syntax and runtime errors.

1. Start IDLE and open the mpg.py file that should be in this folder:
   python/exercises/ch02
2. Press F5 to compile and run the program. Then, enter valid values for miles driven and gallons used. This should display the miles per gallon in the interactive shell.
3. Test the program with invalid entries like spaces or letters. This should cause the program to crash and display error messages for the exceptions that occur.
4. Modify this program so the result is rounded to just one decimal place. Then, test this change.
5. Modify this program so the argument of the round() function is the arithmetic expression in the previous statement. Then, test this change.
6. Modify this program so it gets the cost of a gallon of gas as an entry from the user. Then, calculate the total gas cost and the cost per mile, and display the results on the console as shown above.

Exercise 2-2 Modify the Test Scores program

In this exercise, you'll modify the Test Scores program in figure 2-16. When you're finished, the program will display the three scores entered by the user in a single line, as shown in this console:

```
Enter 3 test scores
======================
Enter test score: 75
Enter test score: 85
Enter test score: 95
======================
Your Scores:   75 85 95
Total Score:   255
Average Score: 85
```

If you have any problems when you test your changes, please refer to figure 1-9 of the last chapter, which shows how to fix syntax and runtime errors.

1. Start IDLE and open the test_scores.py file that should be in this folder:
   `python/exercises/ch02`
2. Press F5 to compile and run the program. Then, enter valid values for the three scores. This should display the results in the interactive shell.
3. Modify this program so it saves the three scores that are entered in variables named score1, score2, and score3. Then, add these scores to the total_score variable, instead of adding the entries to the total_score variable without ever saving them.
4. Display the scores that have been entered before the other results, as shown above.

Exercise 2-3 Create a simple program

Copying and modifying an existing program is often a good way to start a new program. So in this exercise, you'll copy and modify the Miles Per Gallon program so it gets the length and width of a rectangle from the user, calculates the area and the perimeter of the rectangle, and displays the results in the console like this:

```
The Area and Perimeter program

Please enter the length: 25
Please enter the width:  10

Area = 250
Perimeter = 70

Bye!
```

1. Start IDLE and open the mpg_model.py file that is in this folder:
   `python/exercises/ch02`

   Then, before you do anything else use the File→Save As command to save the file as rectangle.py.
2. Modify the code for this program so it works for the new program. Remember that the area of a rectangle is just length times width, and the perimeter is 2 times length plus 2 times width.