Chapter 6 (first part): Files and Exceptions

These slides have been prepared based on the book titled "Starting Out with Python" by Tony Gaddis.

Lecturer: F. Kuzey Edes-Huyal

Bahcesehir University

November 14, 2023

Introduction to File Input and Output

The programs you have written so far require the user to reenter data each time the program runs, because data stored in RAM (referenced by variables) disappears once the program stops running. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. Data stored in a file can be retrieved and used at a later time.

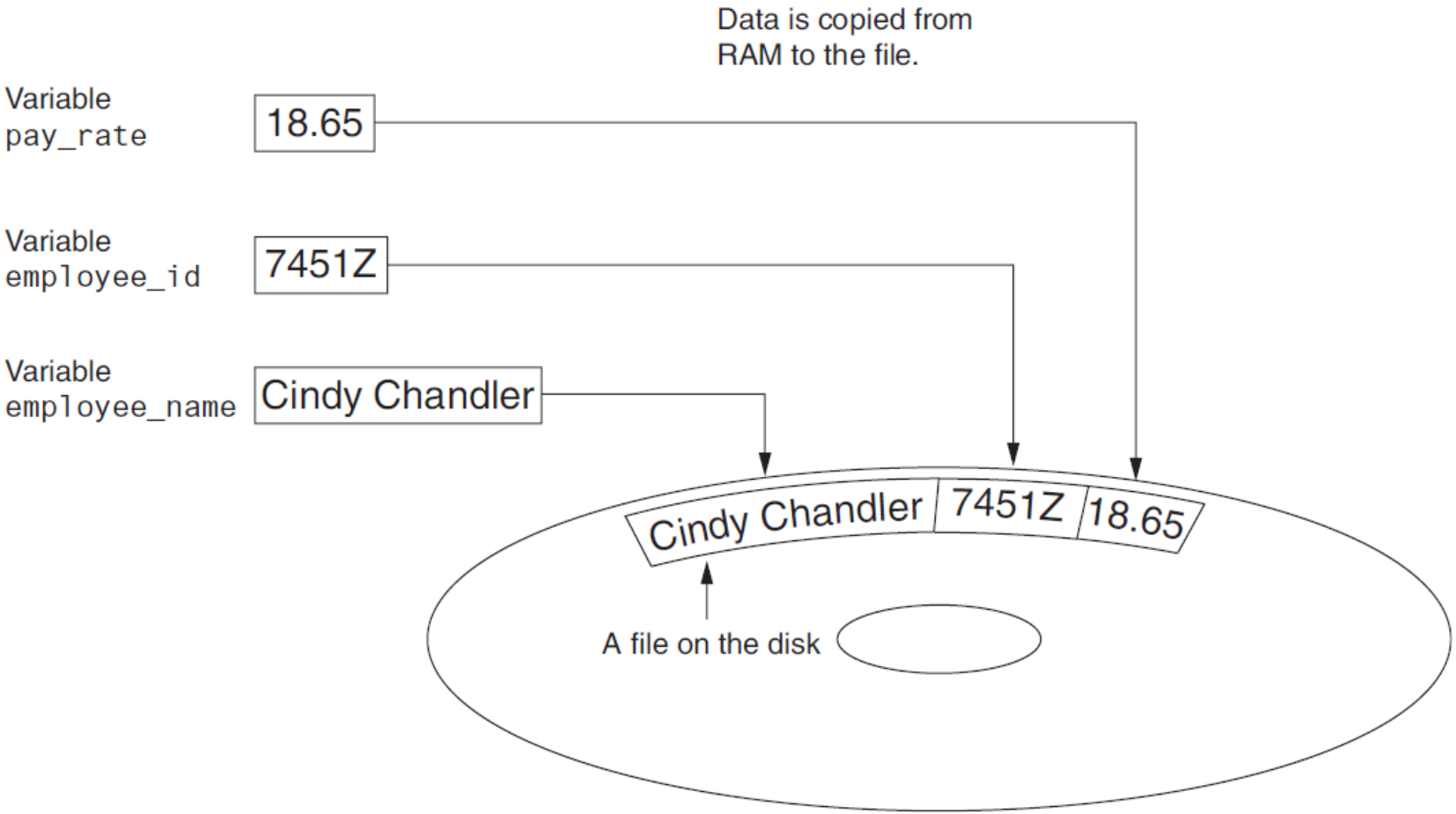
Most of the commercial software packages that you use on a day-to-day basis store data in files. The following are a few examples:

- Word processors
- Image editors
- Spreadsheets
- Games
- Web browers

Programs that are used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company's products in files, accounting systems keep data about a company's financial operations in files, and so on.

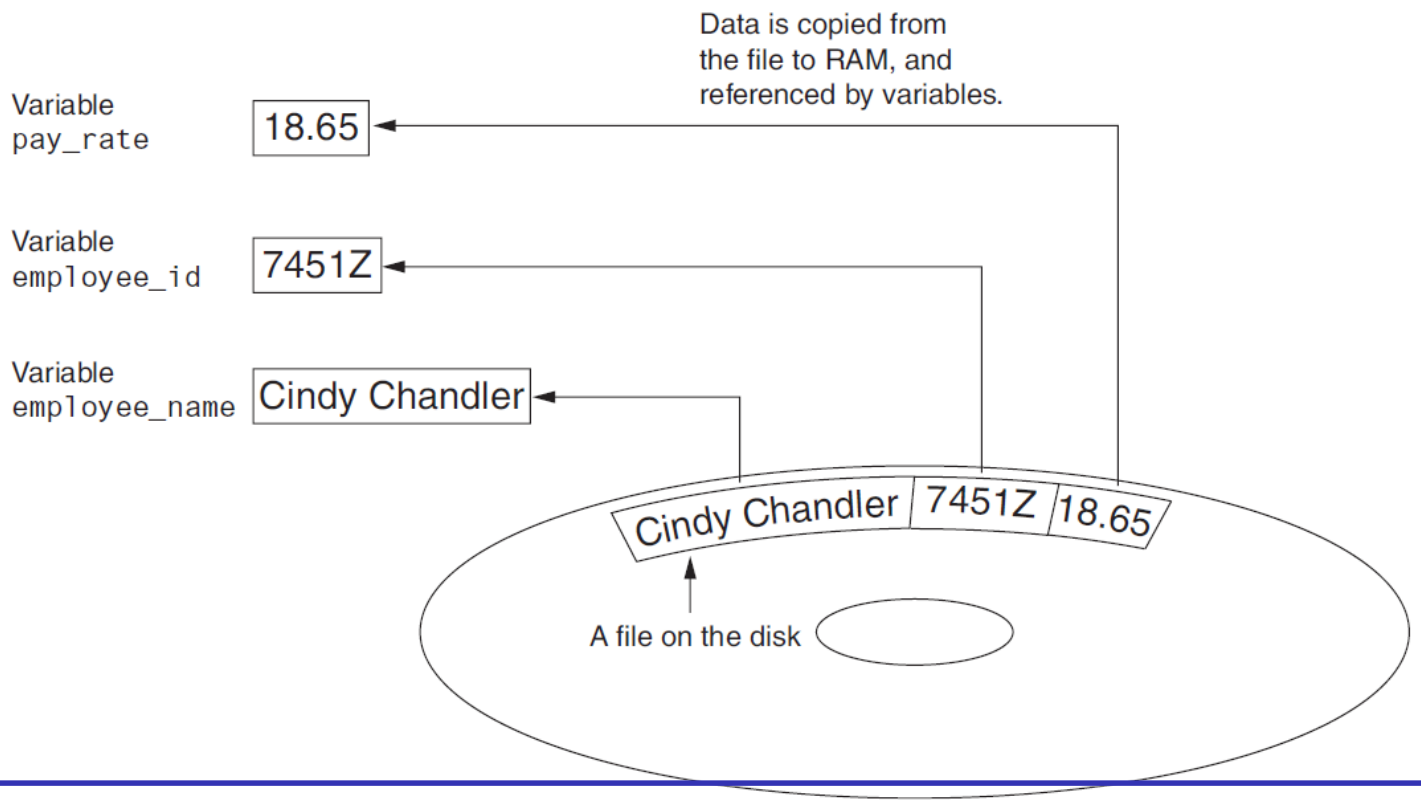
Programmers usually refer to the process of saving data in a file as "writing data" to the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 6-1. The term output file is used to describe a file that data is written to. It is called an output file because the program stores output in it.

Figure 6-1 Writing data to a file



The process of retrieving data from a file is known as "reading data" from the file. When a piece of data is read from a file, it is copied from the file into RAM and referenced by a variable. Figure 6-2 illustrates this. The term input file is used to describe a file from which data is read. It is called an input file because the program gets input from the file.

Figure 6-2 Reading data from a file



This chapter discusses how to write data to files and read data from files. There are always three steps that must be taken when a file is used by a program.

Open the file. Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file.

Process the file. In this step, data is either written to the file (if it is an output file) or read from the file (if it is an input file).

Close the file. When the program is finished using the file, the file must be closed.Closing a file disconnects the file from the program.

In general, there are two types of files: text and binary. A text file contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A binary file contains data that has not been converted to text. The data that is stored in a binary file is intended only for a program to read. As a consequence, you cannot view the contents of a binary file with a text editor.

Although Python allows you to work both text files and binary files, we will work only with text files in this book. That way, you will be able to use an editor to inspect the files that your programs create.

File Access Methods

Most programming languages provide two different ways to access data stored in a file: sequential access and direct access. When you work with a sequential access file, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all of the data that comes before it—you cannot jump directly to the desired data. This is similar to the way older cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

When you work with a direct access file (which is also known as a random access file), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song that you want to listen to. In this lecture, we will use sequential access files.

Most computer users are accustomed to the fact that files are identified by a filename. For example, when you create a document with a word processor and save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Each operating system has its own rules for naming files. Many systems support the use of filename extensions, which are short sequences of characters that appear at the end of a filename preceded by a period (which is known as a "dot").

The extension usually indicates the type of data stored in the file. For example, the .jpg extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The .txt extension usually indicates that the file contains text. The .doc extension (as well as the .docx extension) usually indicates that the file contains a Microsoft Word document.
In order for a program to work with a file on the computer's disk, the program must create a file object in memory. A **file object** is an object that is associated with a specific file and provides a way for the program to work with that file. In the program, a variable references the file object. This variable is used to carry out any operations that are performed on the file.

Opening a File

You use the `open` function in Python to open a file. The open function creates a file object and associates it with a file on the disk. Here is the general format of how the `open` function is used:

```
file_variable = open(filename, mode)
```

In the general format:

- `file_variable` is the name of the variable that will reference the file object.
- `filename` is a string specifying the name of the file.
- `mode` is a string specifying the mode (reading, writing, etc.) in which the file will be opened. Table 6-1 shows three of the strings that you can use to specify a mode. (There are other, more complex modes. The modes shown in Table 6-1 are the ones we will use in this book.)

Table 6-1 Some of the Python file modes

Mode	Description
`'r'`	Open a file for reading only. The file cannot be changed or written to.
`'w'`	Open a file for writing. If the file already exists, erase its contents. If it does not exist, create it.
`'a'`	Open a file to be written to. All data written to the file will be appended to its end. If the file does not exist, create it.

For example, suppose the file `customers.txt` contains customer data, and we want to open it for reading. Here is an example of how we would call the open function:

```
customer_file = open('customers.txt', 'r')
```

After this statement executes, the file named `customers.txt` will be opened, and the variable `customer_file` will reference a file object that we can use to read data from the file. Suppose we want to create a file named sales.txt and write data to it. Here is an example of how we would call the open function:

```
sales_file = open('sales.txt', 'w')
```

After this statement executes, the file named `sales.txt` will be created, and the variable `sales_file` will reference a file object that we can use to write data to the file.

Specifying the Location of a File

When you pass a file name that does not contain a path as an argument to the open function, the Python interpreter assumes the file's location is the same as that of the program. For example, suppose a program is located in the following folder on a Windows computer:

`C:\User\Blake\Document\Python`

If the program is running and it executes the following statement, the file test.txt is created in the same folder:

```
test_file = open('test.txt', 'w')
```

If you want to open a file in a different location, you can specify a path as well as a filename in the argument that you pass to the open function. If you specify a path in a string literal (particularly on a Windows computer), be sure to prefix the string with the letter **r**. Here is an example:

```
test_file = open(r'C:\Users\Blake\temp\test.txt', 'w')
```

Writing Data to a File

So far in this lecture, you have worked with several of Python's library functions, and you have even written your own functions. Now, we will introduce you to another type of function, which is known as a method. A method is a function that belongs to an object and performs some operation using that object. Once you have opened a file, you use the file object's methods to perform operations on the file.

For example, file objects have a method named write that can be used to write data to a file. Here is the general format of how you call the write method:

```
file_variable.write(string)
```

In the format, `file_variable` is a variable that references a file object, and string is a string that will be written to the file. **The file must be opened for writing (using the 'w' or 'a' mode) or an error will occur.**

Let's assume `customer_file` references a file object, and the file was opened for writing with the 'w' mode. Here is an example of how we would write the string 'Charles Pace' to the file:

```
customer_file.write('Charles Pace')
```

The following code shows another example:

```
name = 'Charles Pace'
customer_file.write(name)
```

The second statement writes the value referenced by the name variable to the file associated with `customer_file`. In this case, it would write the string 'Charles Pace' to the file. (These examples show a string being written to a file, but you can also write numeric values.)

In Python, you use the file object's close method to close a file. For example, the following statement closes the file that is associated with `customer_file`:

```
customer_file.close()
```

Program 6-1 shows a complete Python program that opens an output file, writes data to it, then closes it.

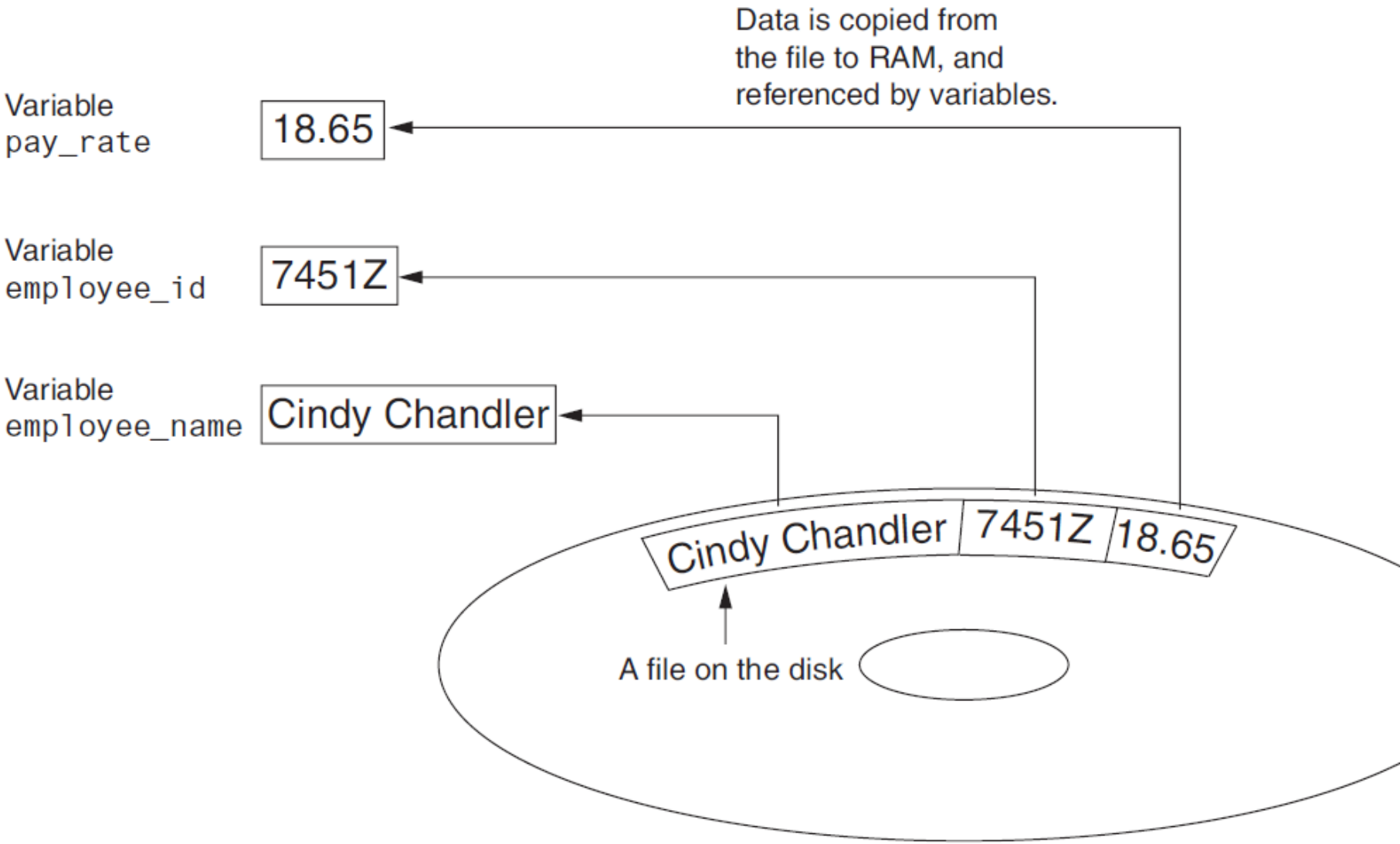
Program 6-1 (file_write.py)

```
1  # This program writes three lines of data
2  # to a file.
3  def main():
4      # Open a file named philosophers.txt.
5      outfile = open('philosophers.txt', 'w')
6
7      # Write the names of three philosphers
8      # to the file.
9      outfile.write('John Locke\n')
10     outfile.write('David Hume\n')
11     outfile.write('Edmund Burke\n')
12
13     # Close the file.
14     outfile.close()
15
16 # Call the main function.
17 if __name__ == '__main__':
18     main()
```

Reading Data From a File

If a file has been opened for reading (using the 'r' mode) you can use the file object's read method to read its entire contents into memory. When you call the read method, it returns the file's contents as a string. For example, Program 6-2 shows how we can use the read method to read the contents of the `philosophers.txt` file we created earlier.

Figure 6-2 Reading data from a file



Although the read method allows you to easily read the entire contents of a file with one statement, many programs need to read and process the items that are stored in a file one at a time. For example, suppose a file contains a series of sales amounts, and you need to write a program that calculates the total of the amounts in the file. The program would read each sale amount from the file and add it to an accumulator.

In Python, you can use the **`readline`** method to read a line from a file. (A line is simply a string of characters that are terminated with a \n.) The method returns the line as a string, including the \n.

Program 6-3 shows how we can use the **`readline`** method to read the of the `philosophers.txt` file, one line at a time.

Program 6-3 (line_read.py)

```
1  # This program reads the contents of the
2  # philosophers.txt file one line at a time.
3  def main():
4      # Open a file named philosophers.txt.
5      infile = open('philosophers.txt', 'r')
6
7      # Read three lines from the file.
8      line1 = infile.readline()
9      line2 = infile.readline()
10     line3 = infile.readline()
```

(program continues)

Program 6-3 *(continued)*

```
11
12     # Close the file.
13     infile.close()
14
15     # Print the data that was read into
16     # memory.
17     print(line1)
18     print(line2)
19     print(line3)
20
21 # Call the main function.
22 if __name__ == '__main__':
23     main()
```

Program Output

```
John Locke
David Hume
Edmund Burke
```

Concatenating a Newline to a String

Program 6-1 wrote three string literals to a file, and each string literal ended with a \n escape sequence. In most cases, the data items that are written to a file are not string literals, but values in memory that are referenced by variables. This would be the case in a program that prompts the user to enter data and then writes that data to a file. When a program writes data that has been entered by the user to a file, it is usually necessary to concatenate a \n escape sequence to the data before writing it. This ensures that each piece of data is written to a separate line in the file. Program 6-4 demonstrates how this is done.

Program 6-4 (write_names.py)

```
1  # This program gets three names from the user
2  # and writes them to a file.
3
4  def main():
5      # Get three names.
6      print('Enter the names of three friends.')
7      name1 = input('Friend #1: ')
8      name2 = input('Friend #2: ')
9      name3 = input('Friend #3: ')
10
11     # Open a file named friends.txt.
12     myfile = open('friends.txt', 'w')
13
14     # Write the names to the file.
15     myfile.write(name1 + '\n')
16     myfile.write(name2 + '\n')
17     myfile.write(name3 + '\n')
18
19     # Close the file.
20     myfile.close()
21     print('The names were written to friends.txt.')
22
23 # Call the main function.
24 if __name__ == '__main__':
25     main()
```

Program Output (with input shown in bold)

```
Enter the names of three friends.
Friend #1: Joe [Enter]
Friend #2: Rose [Enter]
Friend #3: Geri [Enter]
The names were written to friends.txt.
```

Reading a String and Stripping the Newline from It

Sometimes complications are caused by the \n that appears at the end of the strings that are returned from the `readline` method. For example, did you notice in the sample output of Program 6-3 that a blank line is printed after each line of output? This is because each of the strings that are printed in lines 17 through 19 end with a \n escape sequence. When the strings are printed, the \n causes an extra blank line to appear. The \n serves a necessary purpose inside a file: it separates the items that are stored in the file.

However, in many cases, you want to remove the \n from a string after it is read from a file. Each string in Python has a method named `rstrip` that removes, or "strips," specific characters from the end of a string. (It is named rstrip because it strips characters from the right side of a string.) The following code shows an example of how the `rstrip` method can be used.

```
name = 'Joanne Manchester\n'
name = name.rstrip('\n')
```

Program 6-5 (strip_newline.py)

```
1  # This program reads the contents of the
2  # philosophers.txt file one line at a time.
3  def main():
4      # Open a file named philosophers.txt.
5      infile = open('philosophers.txt', 'r')
6
7      # Read three lines from the file
8      line1 = infile.readline()
9      line2 = infile.readline()
10     line3 = infile.readline()
11
12     # Strip the \n from each string.
13     line1 = line1.rstrip('\n')
14     line2 = line2.rstrip('\n')
```

(program continues

Program 6-5 *(continued)*

```
15     line3 = line3.rstrip('\n')
16
17     # Close the file.
18     infile.close()
19
20     # Print the data that was read into memory.
21     print(line1)
22     print(line2)
23     print(line3)
24
25 # Call the main function.
26 if __name__ == '__main__':
27     main()
```

Program Output

```
John Locke
David Hume
Edmund Burke
```

Appending Data to an Existing File

When you use the `'w'` mode to open an output file and a file with the specified filename already exists on the disk, the existing file will be deleted and a new empty file with the same name will be created. Sometimes you want to preserve an existing file and append new data to its current contents. Appending data to a file means writing new data to the end of the data that already exists in the file.
In Python, you can use the `'a'` mode to open an output file in append mode, which means the following.

- If the file already exists, it will not be erased. If the file does not exist, it will be created.
- When data is written to the file, it will be written at the end of the file's current contents.

For example, assume the file **friends.txt** contains the following names, each in a separate line:
Joe
Rose
Geri
The following code opens the file and appends additional data to its existing contents.

```
myfile = open('friends.txt', 'a')
myfile.write('Matt\n')
myfile.write('Chris\n')
myfile.write('Suze\n')
myfile.close()
```

After this program runs, the file friends.txt will contain the following data: Joe
Rose
Geri
Matt
Chris
Suze

Strings can be written directly to a file with the **write** method, **but numbers must be converted to strings before they can be written**. Python has a built-in function named `str` that converts a value to a string. For example, assuming the variable num is assigned the value 99, the expression `str(num)` will return the string '99'. Program 6-6 shows an example of how you can use the `str` function to convert a number to a string, and write the resulting string to a file.

```
1  # This program demonstrates how numbers
2  # must be converted to strings before they
3  # are written to a text file.
4
5  def main():
6      # Open a file for writing.
7      outfile = open('numbers.txt', 'w')
8
9      # Get three numbers from the user.
10     num1 = int(input('Enter a number: '))
11     num2 = int(input('Enter another number: '))
12     num3 = int(input('Enter another number: '))
13
14     # Write the numbers to the file.
15     outfile.write(str(num1) + '\n')
16     outfile.write(str(num2) + '\n')
17     outfile.write(str(num3) + '\n')
18
19     # Close the file.
20     outfile.close()
21     print('Data written to numbers.txt')
22
23 # Call the main function.
24 if __name__ == '__main__':
25     main()
```

(program conti

Program 6-6 *(continued)*

Program Output (with input shown in bold)

```
Enter a number: 22 Enter
Enter another number: 14 Enter
Enter another number: -99 Enter
Data written to numbers.txt
```

When you read numbers from a text file, they are always read as strings. For example, suppose a program uses the following code to read the first line from the numbers.txt file that was created by Program 6-6:

```
1 infile = open('numbers.txt', 'r')
2 value = infile.readline()
3 infile.close()
```

The statement in line 2 uses the `readline` method to read a line from the file. After this statement executes, the value variable will reference the string '22\n'. **This can cause a problem if we intend to perform math with the value variable, because you cannot perform math on strings.** In such a case **you must convert the string to a numeric type.**

Recall from Chapter 2 that Python provides the built-in function `int` to convert a string to an integer, and the built-in function float to convert a string to a floating-point number. For example, we could modify the code previously shown as follows:

```
1 infile = open('numbers.txt', 'r')
2 string_input = infile.readline()
3 value = int(string_input)
4 infile.close()
```

Program 6-7 (read_numbers.py)

```
1  # This program demonstrates how numbers that are
2  # read from a file must be converted from strings
3  # before they are used in a math operation.
4
5  def main():
6      # Open a file for reading.
7      infile = open('numbers.txt', 'r')
8
9      # Read three numbers from the file.
10     num1 = int(infile.readline())
11     num2 = int(infile.readline())
12     num3 = int(infile.readline())
```

(program continues)

Program 6-7 *(continued)*

```
13
14     # Close the file.
15     infile.close()
16
17     # Add the three numbers.
18     total = num1 + num2 + num3
19
20     # Display the numbers and their total.
21     print(f'The numbers are: {num1}, {num2}, {num3}')
22     print(f'Their total is: {total}')
23
24 # Call the main function.
25 if __name__ == '__main__':
26     main()
```

Program Output

```
The numbers are: 22, 14, -99
```

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in Program 6-8. This program gets sales amounts for a series of days from the user and writes those amounts to a file named `sales.txt`. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days. Figure 6-16 shows the contents of the `sales.txt` file containing the data entered by the user in the sample run.

Program 6-8 (write_sales.py)

```
1  # This program prompts the user for sales amounts
2  # and writes those amounts to the sales.txt file.
3
4  def main():
5      # Get the number of days.
6      num_days = int(input('For how many days do ' +
7                           'you have sales? '))
8
9      # Open a new file named sales.txt.
10     sales_file = open('sales.txt', 'w')
11
12     # Get the amount of sales for each day and write
13     # it to the file.
14     for count in range(1, num_days + 1):
15         # Get the sales for a day.
16         sales = float(input(
17             f'Enter the sales for day #{count}: '))
18
19         # Write the sales amount to the file.
20         sales_file.write(f'{sales}\n')
21
22     # Close the file.
23     sales_file.close()
24     print('Data written to sales.txt.')
25
26 # Call the main function.
27 if __name__ == '__main__':
28     main()
```

(program continues)

Program 6-8 *(continued)*

Program Output (with input shown in bold)

```
For how many days do you have sales? 5 Enter
Enter the sales for day #1: 1000.0 Enter
Enter the sales for day #2: 2000.0 Enter
Enter the sales for day #3: 3000.0 Enter
Enter the sales for day #4: 4000.0 Enter
Enter the sales for day #5: 5000.0 Enter
Data written to sales.txt.
```

Reading a File with a Loop and Detecting the End of the File

Quite often, a program must read the contents of a file without knowing the number of items that are stored in the file. For example, the `sales.txt` file that was created by Program 6-8can have any number of items stored in it, because the program asks the user for the number of days for which he or she has sales amounts. If the user enters 5 as the number of days, the program gets 5 sales amounts and writes them to the file. If the user enters 100 as the number of days, the program gets 100 sales amounts and writes them to the file. This presents a problem if you want to write a program that processes all of the items in the file, however many there are. For example, suppose you need to write a program that reads all of the amounts in the `sales.txt` file and calculates their total. You can use a loop to read the items in the file, but you need a way of knowing when the end of the file has been reached.

In Python, the **`readline`** method returns an empty string (”) when it has attempted to read beyond the end of a file. This makes it possible to write a while loop that determines when the end of a file has been reached. Here is the general algorithm, in pseudocode:

```
Open the file
Use readline to read the first line from the file
While the value returned from readline is not an empty
string:
    Process the item that was just read from the file
    Use readline to read the next line from the file.
Close the file
```

```
1  # This program reads all of the values in
2  # the sales.txt file.
3
4  def main():
5      # Open the sales.txt file for reading.
6      sales_file = open('sales.txt', 'r')
7
8      # Read the first line from the file, but
9      # don't convert to a number yet. We still
10     # need to test for an empty string.
11     line = sales_file.readline()
12
13     # As long as an empty string is not returned
14     # from readline, continue processing.
```

(program continues)

Program 6-9 *(continued)*

```
15     while line != '':
16         # Convert line to a float.
17         amount = float(line)
18
19         # Format and display the amount.
20         print(f'{amount:.2f}')
21
22         # Read the next line.
23         line = sales_file.readline()
24
25     # Close the file.
26     sales_file.close()
27
28 # Call the main function.
29 if __name__ == '__main__':
30     main()
```

Program Output

```
1000.00
2000.00
3000.00
4000.00
5000.00
```

Possible Quiz Questions

Kevin is a freelance video producer who makes TV commercials for local businesses. When he makes a commercial, he usually films several short videos. Later, he puts these short videos together to make the final commercial. He has asked you to write the following two programs.

1. A program that allows him to enter the running time (in seconds) of each short video in a project. The running times are saved to a file.
2. A program that reads the contents of the file, displays the running times, and then displays the total running time of all the segments.

Here is the general algorithm for the first program, in pseudocode:

```
Get the number of videos in the project.
Open an output file.
For each video in the project:
    Get the video's running time.
    Write the running time to the file.
Close the file.
```

Program 6-11 shows the code for the first program.

Program 6-11 (save_running_times.py)

```
1  # This program saves a sequence of video running times
2  # to the video_times.txt file.
3
4  def main():
5      # Get the number of videos in the project.
6      num_videos = int(input('How many videos are in the project? '))
7
8      # Open the file to hold the running times.
9      video_file = open('video_times.txt', 'w')
10
11     # Get each video's running time and write
12     # it to the file.
13     print('Enter the running times for each video.')
14     for count in range(1, num_videos + 1):
15         run_time = float(input(f'Video #{count}: '))
16         video_file.write(f'{run_time}\n')
17
18     # Close the file.
19     video_file.close()
20     print('The times have been saved to video_times.txt.')
21
22 # Call the main function.
23 if __name__ == '__main__':
24     main()
```

Program Output (with input shown in bold)

```
How many videos are in the project? 6 Enter
Enter the running times for each video.
Video #1: 24.5 Enter
Video #2: 12.2 Enter
Video #3: 14.6 Enter
Video #4: 20.4 Enter
Video #5: 22.5 Enter
Video #6: 19.3 Enter
The times have been saved to video_times.txt.
```

Here is the general algorithm for the second program:

```
Initialize an accumulator to 0.
Initialize a count variable to 0.
Open the input file.
For each line in the file:
    Convert the line to a floating-point number.
(This is the running time for a video.)
    Add one to the count variable.  (This keeps count
of the number of videos.)
    Display the running time for this video.
    Add the running time to the accumulator.
Close the file.
Display the contents of the accumulator as the total
running time.
```

Program 6-12 shows the code for the second program.

```
# This program the values in the video_times.txt
# file and calculates their total.

def main():
    # Open the video_times.txt file for reading.
    video_file = open('video_times.txt', 'r')

    # Initialize an accumulator to 0.0.
    total = 0.0

    # Initialize a variable to keep count of the videos.
    count = 0

    print('Here are the running times for each video:')

    # Get the values from the file and total them.
    for line in video_file:
        # Convert a line to a float.
        run_time = float(line)

        # Add 1 to the count variable.
        count += 1

        # Display the time.
        print('Video #', count, ': ', run_time, sep='')

        # Add the time to total.
        total += run_time

    # Close the file.
    video_file.close()
```

(program continues)

Program 6-12 *(continued)*

```

    # Display the total of the running times.
    print(f'The total running time is {total} seconds.')

# Call the main function.
if __name__ == '__main__':
    main()
```

Program Output

```
Here are the running times for each video:
Video #1: 24.5
Video #2: 12.2
Video #3: 14.6
Video #4: 20.4
Video #5: 22.5
Video #6: 19.3
The total running time is 113.5 seconds.
```

1. Many financial experts advise that property owners should insure their homes or buildings for at least 80 percent of the amount it would cost to replace the structure. Write a program that asks the user to enter the replacement cost of a building, then displays the minimum amount of insurance he or she should buy for the property.
2. **optional** A nutritionist who works for a fitness club helps members by evaluating their diets. As part of her evaluation, she asks members for the number of fat grams and carbohydrate grams that they consumed in a day. Then, she calculates the number of calories that result from the fat, using the following formula:
   calories from fat = fat grams * 9
   Next, she calculates the number of calories that result from the carbohydrates, using the following formula:
   calories from carbs = carb grams * 4
   The nutritionist asks you to write a program that will make these calculations.