

# Object Oriented Programming II

Dr. Burak Kaleci

May 3, 2019

# Content

Files

Exceptions

# Introduction

- ▶ 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.

# Introduction

- ▶ 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.
- ▶ 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.
- ▶ 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.
- ▶ 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

# Introduction

- ▶ 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.

# Types of Files

- ▶ 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.

# 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 and you cannot jump directly to the desired data.
- ▶ 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.

# File Objects

- ▶ 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.

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.

# Opening a File

- ▶ 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.
- ▶ **Remember, when you use the 'w' mode, you are creating the file on the disk. If a file with the specified name already exists when the file is opened, the contents of the existing file will be deleted.**

# 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.
- ▶ 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')
```
- ▶ The r prefix specifies that the string is a raw string. This causes the Python interpreter to read the backslash characters as literal backslashes.
- ▶ Without the r prefix, the interpreter would assume that the backslash characters were part of escape sequences, and an error would occur.

# Writing Data to a File

- ▶ 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.

# Writing Data to a File

- ▶ 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:

```
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.

# Closing a File

- ▶ Once a program is finished working with a file, it should close the file.
- ▶ Closing a file disconnects the program from the file.
- ▶ In some systems, failure to close an output file can cause a loss of data.
- ▶ The process of closing an output file forces any unsaved data that remains in the buffer to be written to the file.
- ▶ 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()`

# Writing Data to a File

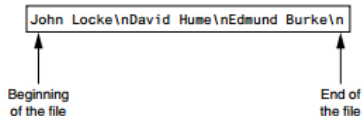
```
1 # This program writes three lines of data
2 # to a file.
3 def main():
4     # Open a file named philosophers.txt.
5     outfile = open('philosophers.txt', 'w')
6
7     # Write the names of three philosophers
8     # to the file.
9     outfile.write('John Locke\n')
10    outfile.write('David Hume\n')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



- ▶ Line 5 opens the file `philosophers.txt` using the `'w'` mode. (This causes the file to be created and opens it for writing.)
- ▶ It also creates a file object in memory and assigns that object to the `outfile` variable.
- ▶ The statements in lines 9 through 11 write three strings to the file.
- ▶ Line 9 writes the string `'John Locke\n'`, line 10 writes the string `'David Hume\n'`, and line 11 writes the string `'Edmund Burke\n'`.
- ▶ Line 14 closes the file.

# Writing Data to a File

- ▶ After this program runs, the three items shown in Figure will be written to the philosophers.txt file.
- ▶ Notice each of the strings written to the file end with `\n`, which you will recall is the newline escape sequence.
- ▶ The `\n` not only separates the items that are in the file, but also causes each of them to appear in a separate line when viewed in a text editor.





# Reading Data From a File

```
1 #This program reads and displays the contents  
2 # of the philosophers.txt file.  
3 def main():  
4     # Open a file named philosophers.txt.  
5     infile = open('philosophers.txt', 'r')  
6  
7     # Read the file's contents.  
8     file_contents = infile.read()  
9  
10    # Close the file.  
11    infile.close()  
12  
13    # Print the data that was read into  
14    # memory.  
15    print(file_contents)  
16  
17    # Call the main function.  
18    main()
```

- ▶ 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.
- ▶ The statement in line 5 opens the philosophers.txt file for reading, using the 'r' mode.

# Reading Data From a File

```
1 #This program reads and displays the contents
2 # of the philosophers.txt file.
3 def main():
4     # Open a file named philosophers.txt.
5     infile = open('philosophers.txt', 'r')
6
7     # Read the file's contents.
8     file_contents = infile.read()
9
10    # Close the file.
11    infile.close()
12
13    # Print the data that was read into
14    # memory.
15    print(file_contents)
16
17    # Call the main function.
18    main()
```

- ▶ It also creates a file object and assigns the object to the `infile` variable.
- ▶ Line 8 calls the `infile.read` method to read the file's contents. The file's contents are read into memory as a string and assigned to the `file_contents` variable.
- ▶ Then the statement in line 15 prints the string that is referenced by the variable.

# 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`.

# Reading Data From a File

```
1#This program reads the contents of the
2# philosophers.txt file one line at a time.
3def 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   # 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.
22main()
```

- ▶ The statement in line 5 opens the philosophers.txt file for reading, using the 'r' mode.

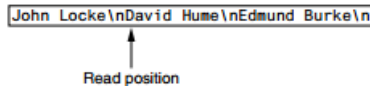
- ▶ It also creates a file object and assigns the object to the infile variable.
- ▶ When a file is opened for reading, a special value known as a read position is internally maintained for that file.
- ▶ A file's read position marks the location of the next item that will be read from the file.
- ▶ Initially, the read position is set to the beginning of the file. After the statement in line 5 executes, the read position for the philosophers.txt file will be positioned as shown in Figure.

John Locke\nDavid Hume\nEdmund Burke\n

Read position

# Reading Data From a File

- ▶ The statement in line 8 calls the `infile.readline` method to read the first line from the file.
- ▶ The line, which is returned as a string, is assigned to the `line1` variable.
- ▶ After this statement executes the `line1` variable will be assigned the string `'John Locke \n'`.
- ▶ In addition, the file's read position will be advanced to the next line in the file, as shown in Figure.



The diagram shows a rectangular box containing the text `John Locke\nDavid Hume\nEdmund Burke\n`. An upward-pointing arrow originates from the text `Read position` below the box and points to the first character of the second line, `D` in `David`. This illustrates that the file's read position has moved to the start of the next line after the first line has been read.

# Reading Data From a File

- ▶ Then the statement in line 9 reads the next line from the file and assigns it to the line2 variable.
- ▶ After this statement executes the line2 variable will reference the string 'David Hume\n'.
- ▶ The file's read position will be advanced to the next line in the file, as shown in Figure

```
John Locke\nDavid Hume\nEdmund Burke\n
```

Read position

# Reading Data From a File

- ▶ Then the statement in line 10 reads the next line from the file and assigns it to the line3 variable.
- ▶ After this statement executes, the line3 variable will reference the string 'Edmund Burke\n'.
- ▶ After this statement executes, the read position will be advanced to the end of the file, as shown in Figure.
- ▶ The statement in line 13 closes the file.
- ▶ The statements in lines 17 through 19 display the contents of the line1, line2, and line3 variables.

```
John Locke\nDavid Hume\nEdmund Burke\n
```

Read position

# Concatenating a Newline to a String

```
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 main()
```

- ▶ 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.



# Concatenating a Newline to a String

```
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 main()
```

- ▶ Lines 7 through 9 prompt the user to enter three names, and those names are assigned to the variables name1, name2, and name3.
- ▶ Line 12 opens a file named friends.txt for writing.
- ▶ Then, lines 15 through 17 write the names entered by the user, each with '\n' concatenated to it.
- ▶ As a result, each name will have the \n escape sequence added to it when written to the file.

# 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.
- ▶ 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.

# Reading a String and Stripping the Newline from It

- ▶ The following code shows an example of how the `rstrip` method can be used:

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

- ▶ The first statement assigns the string `'Joanne Manchester\n'` to the `name` variable.
- ▶ Notice the string ends with the `\n` escape sequence.
- ▶ The second statement calls the `name.rstrip('\n')` method.
- ▶ The method returns a copy of the `name` string without the trailing `\n`.
- ▶ This string is assigned back to the `name` variable.
- ▶ The result is that the trailing `\n` is stripped away from the `name` string.

# Reading a String and Stripping the Newline from It

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

# 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.

# Appending Data to an Existing File

- ▶ For example, assume the file friends.txt contains the following names, each in a separate line:

Joe  
Rose

- ▶ 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.close()
```

- ▶ After this program runs, the file friends.txt will contain the following data:

Joe  
Rose  
Matt  
Chris

# Writing and Reading Numeric Data

```
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 main()
```

- ▶ The statement in line 7 opens the file `numbers.txt` for writing.
- ▶ Then the statements in lines 10 through 12 prompt the user to enter three numbers, which are assigned to the variables `num1`, `num2`, and `num3`.
- ▶ Strings can be written directly to a file with the `write` method, but numbers must be converted to strings before they can be written.

# Writing and Reading Numeric Data

```
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 main()
```

- ▶ Take a closer look at the statement in line 15, which writes the value referenced by num1 to the file:  
`outfile.write(str(num1) + '\n')`
- ▶ The expression `str(num1) + '\n'` converts the value referenced by num1 to a string and concatenates the `\n` escape sequence to the string.
- ▶ In the program's sample run, the user entered 22 as the first number, so this expression produces the string `'22\n'`. As a result, the string `'22\n'` is written to the file.



# Writing and Reading Numeric Data

```
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())
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('The numbers are:', num1, num2, num3)
22    print('Their total is:', total)
23
24 # Call the main function.
25 main()
```

- ▶ When you read numbers from a text file, they are always read as strings.
- ▶ 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 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.

# Using Loops to Process Files

```
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('Enter the sales for day #' +
17                             str(count) + ': '))
18
19        # Write the sales amount to the file.
20        sales_file.write(str(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 main()
```

- ▶ 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.
- ▶ 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.

# 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 can 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.

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

- ▶ 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

# Reading a File with a Loop and Detecting the End of 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.
15    while line != '':
16        # Convert line to a float.
17        amount = float(line)
18
19        # Format and display the amount.
20        print(format(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    main()
```

# Using Python's for Loop to Read Lines

- ▶ In the previous example, you saw how the readline method returns an empty string when the end of the file has been reached.
- ▶ Most programming languages provide a similar technique for detecting the end of a file.
- ▶ The Python language also allows you to write a for loop that automatically reads the lines in a file without testing for any special condition that signals the end of the file.
- ▶ The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached.

# Using Python's for Loop to Read Lines

- ▶ When you simply want to read the lines in a file, one after the other, this technique is simpler and more elegant than writing a while loop that explicitly tests for an end of the file condition.
- ▶ Here is the general format of the loop:  
    for variable in file\_object:  
        statement  
        statement  
        etc.
- ▶ In the general format, variable is the name of a variable, and file\_object is a variable that references a file object.
- ▶ The loop will iterate once for each line in the file.
- ▶ The first time the loop iterates, variable will reference the first line in the file (as a string), the second time the loop iterates, variable will reference the second line, and so forth.

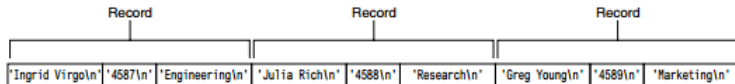
# Using Python's for Loop to Read Lines

```
1 # This program uses the for loop to read
2 # all of the values in 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 all the lines from the file.
9     for line in sales_file:
10         # Convert line to a float.
11         amount = float(line)
12         # Format and display the amount.
13         print(format(amount, '.2f'))
14
15     # Close the file.
16     sales_file.close()
17
18 # Call the main function.
19 main()
```



# Processing Records

- ▶ When data is written to a file, it is often organized into records and fields.
- ▶ A record is a complete set of data that describes one item, and a field is a single piece of data within a record.
- ▶ For example, suppose we want to store data about employees in a file. The file will contain a record for each employee.
- ▶ Each record will be a collection of fields, such as name, ID number, and department.
- ▶ Each time you write a record to a sequential access file, you write the fields that make up the record, one after the other.
- ▶ For example, Figure shows a file that contains three employee records. Each record consists of the employee's name, ID number, and department.



# Processing Records

```
1 # This program gets employee data from the user and
2 # saves it as records in the employee.txt file.
3
4 def main():
5     # Get the number of employee records to create.
6     num_emps = int(input('How many employee records ' +
7                           'do you want to create? '))
8
9     # Open a file for writing.
10    emp_file = open('employees.txt', 'w')
11
12    # Get each employee's data and write it to
13    # the file.
14    for count in range(1, num_emps + 1):
15        # Get the data for an employee.
16        print('Enter data for employee #', count, sep='')
17        name = input('Name: ')
18        id_num = input('ID number: ')
19        dept = input('Department: ')
20
21        # Write the data as a record to the file.
22        emp_file.write(name + '\n')
23        emp_file.write(id_num + '\n')
24        emp_file.write(dept + '\n')
25
26        # Display a blank line.
27        print()
28
29    # Close the file.
30    emp_file.close()
31    print('Employee records written to employees.txt.')
32
33 # Call the main function.
34 main()
```

- ▶ The statement in lines 6 and 7 prompts the user for the number of employee records that he or she wants to create.
- ▶ Inside the loop, in lines 17 through 19, the program gets an employee's name, ID number, and department.
- ▶ These three items, which together make an employee record, are written to the file in lines 22 through 24.
- ▶ The loop iterates once for each employee record

# Processing Records

```
1 # This program displays the records that are
2 # in the employees.txt file.
3
4 def main():
5     # Open the employees.txt file.
6     emp_file = open('employees.txt', 'r')
7
8     # Read the first line from the file, which is
9     # the name field of the first record.
10    name = emp_file.readline()
11
12    # If a field was read, continue processing.
13    while name != '':
14        # Read the ID number field.
15        id_num = emp_file.readline()
16
17        # Read the department field.
18        dept = emp_file.readline()
19
20        # Strip the newlines from the fields.
21        name = name.rstrip('\n')
22        id_num = id_num.rstrip('\n')
23        dept = dept.rstrip('\n')
24
25        # Display the record.
26        print('Name:', name)
27        print('ID:', id_num)
28        print('Dept:', dept)
29        print()
30
31        # Read the name field of the next record.
32        name = emp_file.readline()
33
34    # Close the file.
35    emp_file.close()
36
37    # Call the main function.
38    main()
```

- ▶ When we read a record from a sequential access file, we read the data for each field, one after the other, until we have read the complete record.
- ▶ This program opens the file in line 6, then in line 10 reads the first field of the first record. This will be the first employee's name.
- ▶ The while loop in line 13 tests the value to determine whether it is an empty string.
- ▶ If it is not, then the loop iterates. Inside the loop, the program reads the record's second and third fields (the employee's ID number and department), and displays them.
- ▶ Then, in line 32 the first field of the next record (the next employee's name) is read. The loop starts over and this process continues until there are no more records to read.

# An example

- ▶ Midnight Coffee Roasters, Inc. is a small company that imports raw coffee beans from around the world and roasts them to create a variety of gourmet coffees.
- ▶ Julie, the owner of the company, has asked you to write a series of programs that she can use to manage her inventory.
- ▶ After speaking with her, you have determined that a file is needed to keep inventory records.
- ▶ Each record should have two fields to hold the following data:
  - ▶ Description. A string containing the name of the coffee.
  - ▶ Quantity in inventory. The number of pounds in inventory, as a floating-point number
- ▶ Your first job is to write a program that can be used to add records to the file.
- ▶ Note that you have to open the output file in append mode. Each time the program is executed, the new records will be added to the file's existing contents

## add\_coffee\_record

```
1 | # This program adds coffee inventory records to
2 | # the coffee.txt file.
3
4 def main():
5     # Create a variable to control the loop.
6     another = 'y'
7
8     # Open the coffee.txt file in append mode.
9     coffee_file = open('coffee.txt', 'a')
10
11     # Add records to the file.
12     while another == 'y' or another == 'Y':
13         # Get the coffee record data.
14         print('Enter the following coffee data:')
15         descr = input('Description: ')
16         qty = int(input('Quantity (in pounds): '))
17
18         # Append the data to the file.
19         coffee_file.write(descr + '\n')
20         coffee_file.write(str(qty) + '\n')
21
22         # Determine whether the user wants to add
23         # another record to the file.
24         print('Do you want to add another record?')
25         another = input('Y = yes, anything else = no: ')
26
27     # Close the file.
28     coffee_file.close()
29     print('Data appended to coffee.txt.')
30
31 # Call the main function.
32 main()
```

## show\_coffee\_record

```
1 # This program displays the records in the
2 # coffee.txt file.
3
4 def main():
5     # Open the coffee.txt file.
6     coffee_file = open('coffee.txt', 'r')
7
8     # Read the first record's description field.
9     descr = coffee_file.readline()
10
11     # Read the rest of the file.
12     while descr != '':
13         # Read the quantity field.
14         qty = float(coffee_file.readline())
15
16         # Strip the \n from the description.
17         descr = descr.rstrip('\n')
18
19         # Display the record.
20         print('Description:', descr)
21         print('Quantity:', qty)
22
23         # Read the next description.
24         descr = coffee_file.readline()
25
26     # Close the file.
27     coffee_file.close()
28
29 # Call the main function.
30 main()
```

## search\_coffee\_record

```
1 # This program allows the user to search the coffee.txt file for records matching a description.
2
3 def main():
4     # Create a bool variable to use as a flag.
5     found = False
6
7     # Get the search value.
8     search = input('Enter a description to search for: ')
9
10    # Open the coffee.txt file.
11    coffee_file = open('coffee.txt', 'r')
12
13    # Read the first record's description field.
14    descr = coffee_file.readline()
15
16    # Read the rest of the file.
17    while descr != '':
18        # Read the quantity field.
19        qty = float(coffee_file.readline())
20
21        # Strip the \n from the description.
22        descr = descr.rstrip('\n')
23
24        # Determine whether this record matches the search value.
25        if descr == search:
26            # Display the record.
27            print('Description:', descr)
28            print('Quantity:', qty)
29            print()
30            # Set the found flag to True.
31            found = True
32
33        # Read the next description.
34        descr = coffee_file.readline()
35
36    # Close the file.
37    coffee_file.close()
38
39    # If the search value was not found in the file, display a message.
40    if not found:
41        print('That item was not found in the file.')
42
43    # Call the main function.
44    main()
```

# Modifying Records

- ▶ To modify a record in a sequential file, you must create a second temporary file.
- ▶ You copy all of the original file's records to the temporary file, but when you get to the record that is to be modified, you do not write its old contents to the temporary file.
- ▶ Instead, you write its new modified values to the temporary file. Then, you finish copying any remaining records from the original file to the temporary file.
- ▶ The temporary file then takes the place of the original file.
- ▶ You delete the original file and rename the temporary file, giving it the name that the original file had on the computer's disk.



# Modifying Records

- ▶ Notice at the end of the algorithm you delete the original file then rename the temporary file.
- ▶ The Python standard library's `os` module provides a function named `remove`, that deletes a file on the disk.
- ▶ You simply pass the name of the file as an argument to the function.
- ▶ Here is an example of how you would delete a file named `coffee.txt`:
- ▶ The `os` module also provides a function named `rename`, that renames a file.
- ▶ Here is an example of how you would use it to rename the file `temp.txt` to `coffee.txt`:

```
remove('coffee.txt')
```

```
rename('temp.txt', 'coffee.txt')
```

# Modifying Records

```
1 # This program allows the user to modify the quantity in a record in the coffee.txt file.
2
3 import os # Needed for the remove and rename functions
4
5 def main():
6
7     # Get the search value and the new quantity.
8     search = input('Enter a description to search for: ')
9     new_qty = int(input('Enter the new quantity: '))
10
11     coffee_file = open('coffee.txt', 'r') # Open the original coffee.txt file.
12     temp_file = open('temp.txt', 'w') # Open the temporary file.
13
14     descr = coffee_file.readline() # Read the first record's description field.
15
16     while descr != '':
17         # Read the quantity field.
18         qty = float(coffee_file.readline())
19
20         # Strip the \n from the description.
21         descr = descr.rstrip('\n')
22
23         if descr == search: # Write the modified record to the temp file.
24             temp_file.write(descr + '\n')
25             temp_file.write(str(new_qty) + '\n')
26
27         else: # Write the original record to the temp file.
28             temp_file.write(descr + '\n')
29             temp_file.write(str(qty) + '\n')
30
31     descr = coffee_file.readline() # Read the next description.
32
33     # Close the coffee file and the temporary file.
34     coffee_file.close()
35     temp_file.close()
36
37     os.remove('coffee.txt') # Delete the original coffee.txt file.
38     os.rename('temp.txt', 'coffee.txt') # Rename the temporary file.
39
40 # Call the main function.
41 main()
```

# Deleting Records

- ▶ Your last task is to write a program that Julie can use to delete records from the coffee.txt file.
- ▶ Like the process of modifying a record, the process of deleting a record from a sequential access file requires that you create a second temporary file.
- ▶ You copy all of the original file's records to the temporary file, except for the record that is to be deleted.
- ▶ The temporary file then takes the place of the original file.
- ▶ You delete the original file and rename the temporary file, giving it the name that the original file had on the computer's disk.

# Deleting Records

```
1 # This program allows the user to delete a record in the coffee.txt file.
2
3 import os # Needed for the remove and rename functions
4
5 def main():
6
7     # Get the coffee to delete.
8     search = input('Which coffee do you want to delete? ')
9
10    coffee_file = open('coffee.txt', 'r') # Open the original coffee.txt file.
11    temp_file = open('temp.txt', 'w') # Open the temporary file.
12
13
14    descr = coffee_file.readline() # Read the first record's description field.
15
16    while descr != '':
17        # Read the quantity field.
18        qty = float(coffee_file.readline())
19
20        # Strip the \n from the description.
21        descr = descr.rstrip('\n')
22
23        # If this is not the record to delete, then write it to the temporary file.
24        if descr != search:
25            # Write the record to the temp file.
26            temp_file.write(descr + '\n')
27            temp_file.write(str(qty) + '\n')
28
29        # Read the next description.
30        descr = coffee_file.readline()
31
32    # Close the coffee file and the temporary file.
33    coffee_file.close()
34    temp_file.close()
35
36    os.remove('coffee.txt') # Delete the original coffee.txt file.
37    os.rename('temp.txt', 'coffee.txt') # Rename the temporary file.
38
39    # Call the main function.
40    main()
```

# Introduction

```
1 # This program divides a number by another number.
2
3 def main():
4     # Get two numbers.
5     num1 = int(input('Enter a number: '))
6     num2 = int(input('Enter another number: '))
7
8     # Divide num1 by num2 and display the result.
9     result = num1 / num2
10    print(num1, 'divided by', num2, 'is', result)
11
12 # Call the main function.
13 main()
```

- ▶ An exception is an error that occurs while a program is running.
- ▶ In most cases, an exception causes a program to abruptly halt.

- ▶ For example, the program gets two numbers from the user then divides the first number by the second number.
- ▶ In the sample running of the program, however, an exception occurred because the user entered 0 as the second number.
- ▶ Division by 0 causes an exception because it is mathematically impossible.

# Introduction

- ▶ The lengthy error message that is shown in the sample run is called a traceback.
- ▶ The traceback gives information regarding the line number(s) that caused the exception.
- ▶ The last line of the error message shows the name of the exception that was raised (`ZeroDivisionError`) and a brief description of the error that caused the exception to be raised (integer division or modulo by zero).

```
Traceback (most recent call last):  
  File "C:\Python\division.py," line 13, in <module>  
    main()  
  File "C:\Python\division.py," line 9, in main  
    result = num1 / num2  
ZeroDivisionError: integer division or modulo by zero
```

# Introduction

```
1 # This program divides a number by another number.
2
3 def main():
4     # Get two numbers.
5     num1 = int(input('Enter a number: '))
6     num2 = int(input('Enter another number: '))
7
8     # If num2 is not 0, divide num1 by num2
9     # and display the result.
10    if num2 != 0:
11        result = num1 / num2
12        print(num1, 'divided by', num2, 'is', result)
13    else:
14        print('Cannot divide by zero.')
15
16 # Call the main function.
17 main()
```

- ▶ You can prevent many exceptions from being raised by carefully coding your program.
- ▶ For example, the program shows how division by 0 can be prevented with a simple if statement.
- ▶ Rather than allowing the exception to be raised, the program tests the value of num2, and displays an error message if the value is 0.
- ▶ This is an example of gracefully avoiding an exception.

# Introduction

```
1 # This program calculates gross pay.
2
3 def main():
4     # Get the number of hours worked.
5     hours = int(input('How many hours did you work? '))
6
7     # Get the hourly pay rate.
8     pay_rate = float(input('Enter your hourly pay rate: '))
9
10    # Calculate the gross pay.
11    gross_pay = hours * pay_rate
12
13    # Display the gross pay.
14    print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
15
16 # Call the main function.
17 main()
```

- ▶ Some exceptions cannot be avoided regardless of how carefully you write your program.
- ▶ For example, the program calculates gross pay. It prompts the user to enter the number of hours worked and the hourly pay rate.
- ▶ It gets the user's gross pay by multiplying these two numbers and displays that value on the screen.



# Introduction

- ▶ Look at the sample running of the program. An exception occurred because the user entered the string 'forty' instead of the number 40 when prompted for the number of hours worked.
- ▶ Because the string 'forty' cannot be converted to an integer, the `int()` function raised an exception in line 5, and the program halted.
- ▶ Look carefully at the last line of the traceback message, and you will see that the name of the exception is `ValueError`, and its description is: `invalid literal for int() with base 10: 'forty'`.

## Program Output (with input shown in bold)

```
How many hours did you work? forty 
Traceback (most recent call last):
  File "C:\Users\Tony\Documents\Python\Source
Code\Chapter 06\gross_pay1.py", line 17, in <module>
    main()
  File "C:\Users\Tony\Documents\Python\Source
Code\Chapter 06\gross_pay1.py", line 5, in main
    hours = int(input('How many hours did you work? '))
ValueError: invalid literal for int() with base 10: 'forty'
```

# Introduction

- ▶ Python, like most modern programming languages, allows you to write code that responds to exceptions when they are raised, and prevents the program from abruptly crashing.
- ▶ Such code is called an exception handler and is written with the try/except statement.
- ▶ There are several ways to write a try/except statement, but the following general format shows the simplest variation:

try:

statement

statement

etc.

except ExceptionName:

statement

statement

etc.

# Introduction

- ▶ First, the key word `try` appears, followed by a colon.
- ▶ Next, a code block appears which we will refer to as the try suite.
- ▶ The try suite is one or more statements that can potentially raise an exception.
- ▶ After the try suite, an `except` clause appears.
- ▶ The `except` clause begins with the key word `except`, optionally followed by the name of an exception, and ending with a colon.
- ▶ Beginning on the next line is a block of statements that we will refer to as a handler.

# Introduction

- ▶ When the try/except statement executes, the statements in the try suite begin to execute.
- ▶ The following describes what happens next:
  - ▶ If a statement in the try suite raises an exception that is specified by the ExceptionName in an except clause, then the handler that immediately follows the except clause executes. Then, the program resumes execution with the statement immediately following the try/except statement.
  - ▶ If a statement in the try suite raises an exception that is not specified by the ExceptionName in an except clause, then the program will halt with a traceback error message.
  - ▶ If the statements in the try suite execute without raising an exception, then any except clauses and handlers in the statement are skipped, and the program resumes execution with the statement immediately following the try/except statement.

# Introduction

```
1 # This program calculates gross pay.
2
3 def main():
4     try:
5         # Get the number of hours worked.
6         hours = int(input('How many hours did you work? '))
7
8         # Get the hourly pay rate.
9         pay_rate = float(input('Enter your hourly pay rate: '))
10
11        # Calculate the gross pay.
12        gross_pay = hours * pay_rate
13
14        # Display the gross pay.
15        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
16    except ValueError:
17        print('ERROR: Hours worked and hourly pay rate must')
18        print('be valid numbers.')
```

- ▶ Let's look at what happened in the sample run. The statement in line 6 prompts the user to enter the number of hours worked, and the user enters the string 'forty'.



- ▶ Because the string 'forty' cannot be converted to an integer, the `int()` function raises a `ValueError` exception.
- ▶ As a result, the program jumps immediately out of the try suite to the `except ValueError` clause in line 16 and begins executing the handler block that begins in line 17.

# Introduction

```
4 def main():
5     # Get the name of a file.
6     filename = input('Enter a filename: ')
7
8     # Open the file.
9     infile = open(filename, 'r')
10
11    # Read the file's contents.
12    contents = infile.read()
13
14    # Display the file's contents.
15    print(contents)
16
17    # Close the file.
18    infile.close()
19
20 # Call the main function.
21 main()
```

## Program Output (with input shown in bold)

```
Enter a filename: bad_file.txt  Enter
Traceback (most recent call last):
File "C:\Python\display_file.py," line 21, in <module>
main()
File "C:\Python\display_file.py," line 9, in main
infile = open(filename, 'r')
IOError: [Errno 2] No such file or directory: 'bad_file.txt'
```

- ▶ This program, which does not use exception handling, gets the name of a file from the user then displays the contents of the file.
- ▶ The program works as long as the user enters the name of an existing file. An exception will be raised, however, if the file specified by the user does not exist.
- ▶ The statement in line 9 raised the exception when it called the open function.

# Introduction

```
4 def main():
5     # Get the name of a file.
6     filename = input('Enter a filename: ')
7
8     # Open the file.
9     infile = open(filename, 'r')
10
11    # Read the file's contents.
12    contents = infile.read()
13
14    # Display the file's contents.
15    print(contents)
16
17    # Close the file.
18    infile.close()
19
20 # Call the main function.
21 main()
```

## Program Output (with input shown in bold)

```
Enter a filename: bad_file.txt Enter
Traceback (most recent call last):
File "C:\Python\display_file.py," line 21, in <module>
main()
File "C:\Python\display_file.py," line 9, in main
infile = open(filename, 'r')
IOError: [Errno 2] No such file or directory: 'bad_file.txt'
```

- ▶ Notice in the traceback error message that the name of the exception that occurred is `IOError`.
- ▶ This is an exception that is raised when a file I/O operation fails.
- ▶ You can see in the traceback message that the cause of the error was No such file or directory: 'bad\_file.txt'.

# Introduction

```
1 # This program displays the contents
2 # of a file.
3
4 def main():
5     # Get the name of a file.
6     filename = input('Enter a filename: ')
7
8     try:
9         # Open the file.
10        infile = open(filename, 'r')
11
12        # Read the file's contents.
13        contents = infile.read()
14
15        # Display the file's contents.
16        print(contents)
17
18        # Close the file.
19        infile.close()
20    except IOError:
21        print('An error occurred trying to read')
22        print('the file', filename)
23
24 # Call the main function.
25 main()
```

- ▶ Let's look at what happened in the sample run. When line 6 executed, the user entered `bad_file.txt`, which was assigned to the `filename` variable.
- ▶ Inside the try suite, line 10 attempts to open the file `bad_file.txt`.
- ▶ Because this file does not exist, the statement raises an `IOError` exception.



# Introduction

```
1 # This program displays the contents
2 # of a file.
3
4 def main():
5     # Get the name of a file.
6     filename = input('Enter a filename: ')
7
8     try:
9         # Open the file.
10        infile = open(filename, 'r')
11
12        # Read the file's contents.
13        contents = infile.read()
14
15        # Display the file's contents.
16        print(contents)
17
18        # Close the file.
19        infile.close()
20    except IOError:
21        print('An error occurred trying to read')
22        print('the file', filename)
23
24 # Call the main function.
25 main()
```

- ▶ When this happens, the program exits the try suite, skipping lines 11 through 19.
- ▶ Because the except clause in line 20 specifies the `IOError` exception, the program jumps to the handler that begins in line 21.

# Handling Multiple Exceptions

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20
21        # Print the total.
22        print(format(total, ',.2f'))
23
24    except IOError:
25        print('An error occurred trying to read the file.')
26
27    except ValueError:
28        print('Non-numeric data found in the file.')
29
30    except:
31        print('An error occurred.')
32
33 # Call the main function.
34 main()
```

- ▶ In many cases, the code in a try suite will be capable of throwing more than one type of exception.
- ▶ In such a case, you need to write an except clause for each type of exception that you want to handle.
- ▶ The statement in line 10 can raise an IOError exception if the sales\_data.txt file does not exist.
- ▶ The for loop in line 14 can also raise an IOError exception if it encounters a problem reading data from the file.

# Handling Multiple Exceptions

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20
21        # Print the total.
22        print(format(total, ',.2f'))
23
24    except IOError:
25        print('An error occurred trying to read the file.')
26
27    except ValueError:
28        print('Non-numeric data found in the file.')
29
30    except:
31        print('An error occurred.')
32
33 # Call the main function.
34 main()
```

- ▶ The float function in line 15 can raise a ValueError exception if the line variable references a string that cannot be converted to a floating-point number.
- ▶ Notice the try/except statement has three except clauses.
- ▶ The except clause in line 24 specifies the IOError exception. Its handler in line 25 will execute if an IOError exception is raised.
- ▶ The except clause in line 27 specifies the ValueError exception. Its handler in line 28 will execute if a ValueError exception is raised

# Handling Multiple Exceptions

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20
21        # Print the total.
22        print(format(total, ',.2f'))
23
24    except IOError:
25        print('An error occurred trying to read the file.')
26
27    except ValueError:
28        print('Non-numeric data found in the file.')
29
30    except:
31        print('An error occurred.')
32
33 # Call the main function.
34 main()
```

- ▶ The except clause in line 30 does not list a specific exception. Its handler in line 31 will execute if an exception that is not handled by the other except clauses is raised.
- ▶ If an exception occurs in the try suite, the Python interpreter examines each of the except clauses, from top to bottom, in the try/except statement.
- ▶ When it finds an except clause that specifies a type that matches the type of exception that occurred, it branches to that except clause.
- ▶ If none of the except clauses specifies a type that matches the exception, the interpreter branches to the except clause in line 30.

# Displaying an Exception's Default Error Message

- ▶ When an exception is thrown, an object known as an exception object is created in memory.
- ▶ The exception object usually contains a default error message pertaining to the exception.
- ▶ When you write an except clause, you can optionally assign the exception object to a variable, as shown here:  
`except ValueError as err:`
- ▶ This except clause catches ValueError exceptions.
- ▶ The expression that appears after the except clause specifies that we are assigning the exception object to the variable err. (There is nothing special about the name err. That is simply the name that we have chosen for the examples. You can use any name that you wish.)
- ▶ After doing this, in the exception handler you can pass the err variable to the print function to display the default error message that Python provides for that type of error.

# Displaying an Exception's Default Error Message

```
def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))

        # Get the hourly pay rate.
        pay_rate = float(input('Enter your hourly pay rate: '))

        # Calculate the gross pay.
        gross_pay = hours * pay_rate

        # Display the gross pay.
        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
    except ValueError as err:
        print(err)
```

- ▶ When a `ValueError` exception occurs inside the try suite (lines 5 through 15), the program branches to the except clause in line 16.
- ▶ The expression `ValueError as err` in line 16 causes the resulting exception object to be assigned to a variable named `err`.
- ▶ The statement in line 17 passes the `err` variable to the `print` function, which causes the exception's default error message to be displayed.

# Displaying an Exception's Default Error Message

- ▶ If you want to have just one except clause to catch all the exceptions that are raised in a try suite, you can specify Exception as the type.

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20
21        # Print the total.
22        print(format(total, ',.2f'))
23    except Exception as err:
24        print(err)
25
26 # Call the main function.
27 main()
```

# The else Clause

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20    except Exception as err:
21        print(err)
22    else:
23        # Print the total.
24        print(format(total, ',.2f'))
```

- Here is the general format of a try/except statement with an else clause:

```
try:
    statement
    etc.
except ExceptionName:
    statement
    etc.
else:
    statement
    etc.
```



# The else Clause

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20    except Exception as err:
21        print(err)
22    else:
23        # Print the total.
24        print(format(total, ',.2f'))
```

- ▶ The block of statements that appears after the else clause is known as the else suite.
- ▶ The statements in the else suite are executed after the statements in the try suite, only if no exceptions were raised.
- ▶ If an exception is raised, the else suite is skipped.