M110: Python Programming

Meeting #8

Files and Exceptions

الجامعة العربية المفتوحة
AOU
Arab Open University
Lebanon

Prepared by Dr. Ahmad Mikati

Contents

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 later.
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, 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. 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.

Introduction to File Input and Output

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.

1. **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.

2. **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).

3. 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.

Files: Types and Access Methods

File Types

In general, there are two types of files: text and binary.
Although Python allows you to work both text files and binary files, we will work only with text files.

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. That way, you will be able to use an editor to inspect the files that your programs create.

Files: Types and Access Methods

File Access Methods

Most programming languages provide two different ways to access data stored in a file: sequential access and direct access.

We will use sequential access files. Sequential access files are easy to work
with, and you can use them to gain an understanding of basic file operations.

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 should read all 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.

Opening a File

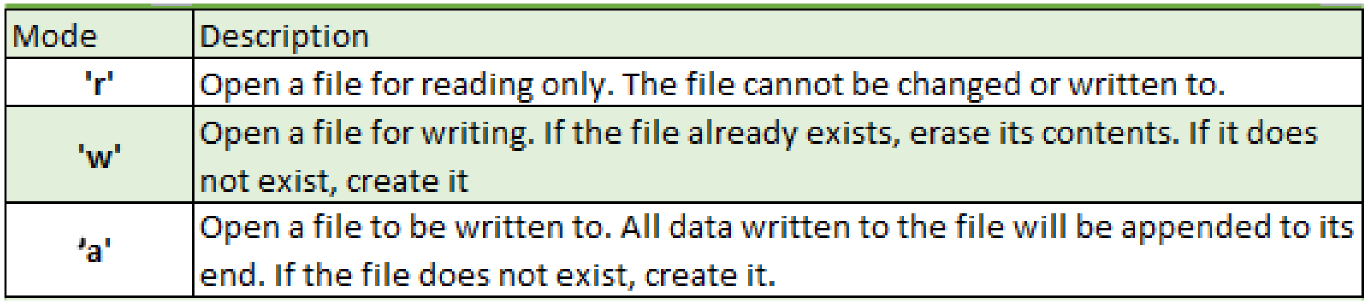
You use the *open* function in Python to open a file.
The open function creates a file object and associates it with a file on the disk. Here is the general format of how the open function is used:

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.

The modes shown in Table 8-1 are the ones we will use.



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.

Table 8-1

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 in the following folder on a Windows computer: **C:\Users\ABM\Documents\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**.
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.
Here is an example: test_file = open(r'C:\Users\ABM\M110\test.txt', 'w')

Writing Data to a File

So far, you have worked with several of Python's library functions, and you have even written your own functions.
Now, we 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. Below is the general format of how you call the *write* method:

file_variable.write(*string*)

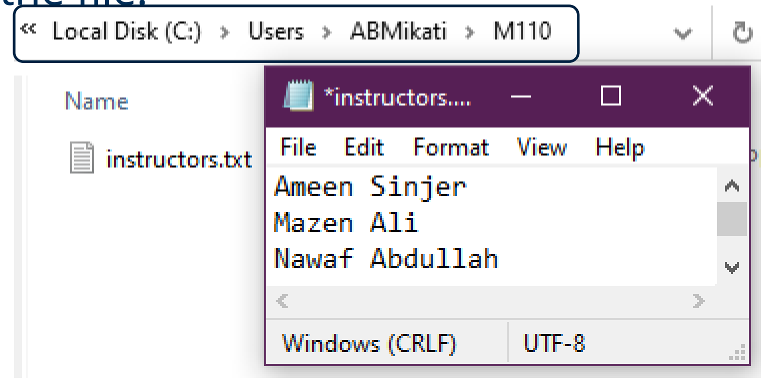
In the format, *file_variable* is a variable that references a file object, and *string* is a string that will be written to the file.
The file must be opened for writing (using the 'w' or 'a' mode) or an error will occur.

AOU
الجامعة العربية المفتوحة
Arab Open University
Lebanon

Writing Data to a File- Example

```
# This program writes three lines of data to a file.
def main():
    # Open a file named instructors.txt.
    myfile = open(r'C:\Users\ABMikati\M110\instructors.txt',
'w’)

    # Write the names of three instructors to the file.
    myfile.write('Ameen Sinjer\n’)
    myfile.write('Mazen Ali\n’)
    myfile.write('Nawaf Abdullah\n’)

    # Close the file.
    myfile.close()

# Call the main function.
main()
```

path


« Local Disk (C:) › Users › ABMikati › M110
Name
instructors.txt
*instructors....
File Edit Format View Help
Windows (CRLF)
UTF-8

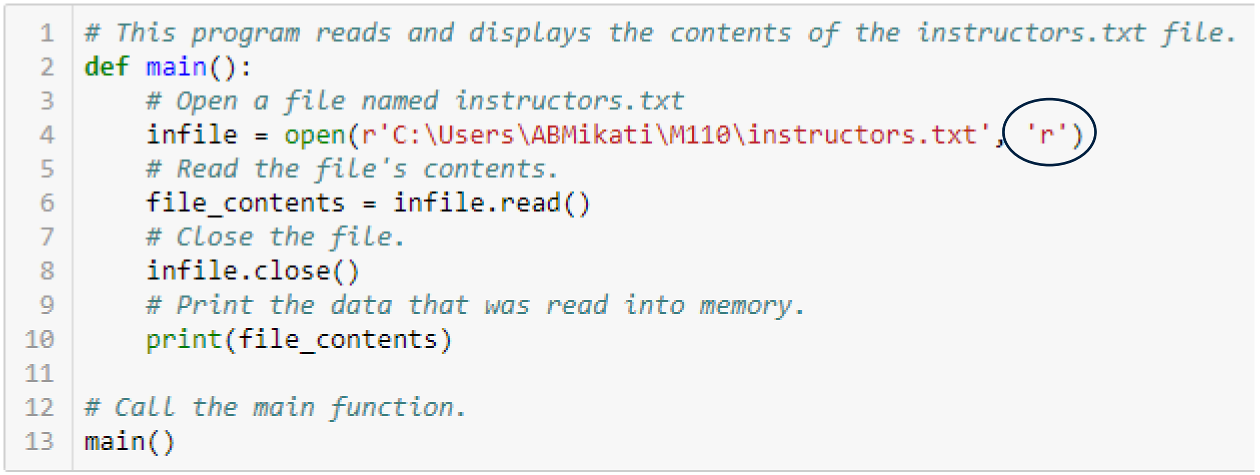
```
Ameen Sinjer
Mazen Ali
Nawaf Abdullah
```

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

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**.



```
 1  # This program reads and displays the contents of the instructors.txt file.
 2  def main():
 3      # Open a file named instructors.txt
 4      infile = open(r'C:\Users\ABMikati\M110\instructors.txt', 'r')
 5      # Read the file's contents.
 6      file_contents = infile.read()
 7      # Close the file.
 8      infile.close()
 9      # Print the data that was read into memory.
10      print(file_contents)
11
12  # Call the main function.
13  main()
```

```
Ameen Sinjer
Mazen Ali
Nawaf Abdullah
```

The statement in line 4 opens the instructors.txt file for reading, using the 'r' mode. It also creates a file object and assigns the object to the infile variable. Line 6 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 10 prints the string that is referenced by the variable.

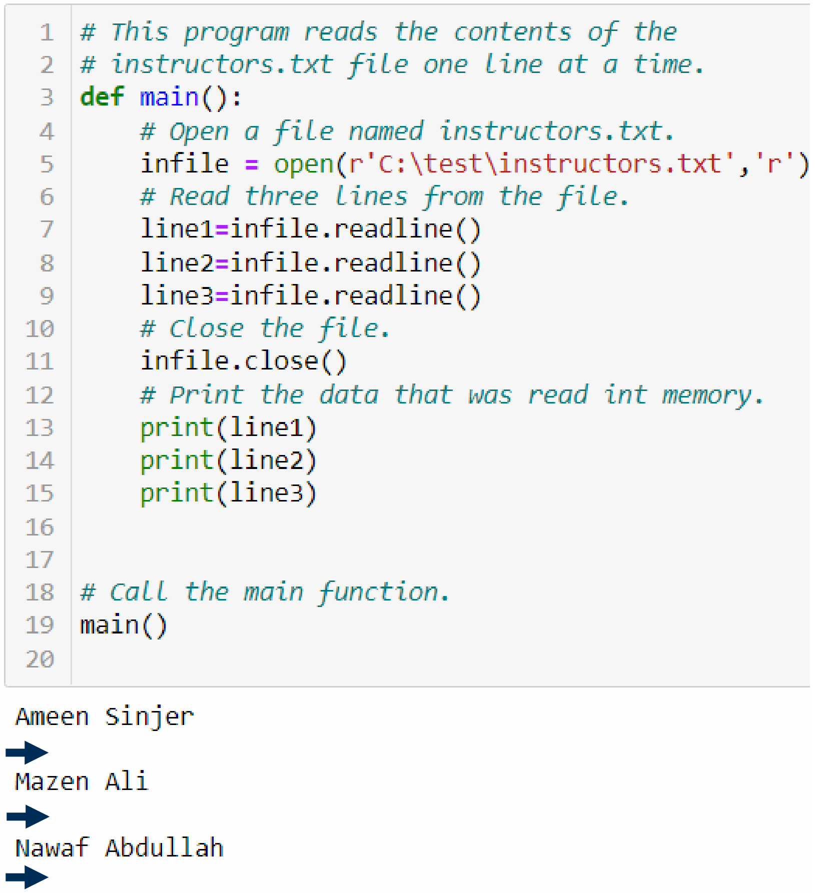
Reading Data From a File

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.

The next Program shows how we can use the *readline* method to read the contents of the instructors.txt file, one line at a time.

Reading Data From a File



```python
# This program reads the contents of the
# instructors.txt file one line at a time.
def main():
    # Open a file named instructors.txt.
    infile = open(r'C:\test\instructors.txt','r')
    # Read three lines from the file.
    line1=infile.readline()
    line2=infile.readline()
    line3=infile.readline()
    # Close the file.
    infile.close()
    # Print the data that was read int memory.
    print(line1)
    print(line2)
    print(line3)


# Call the main function.
main()

```

```
Ameen Sinjer

Mazen Ali

Nawaf Abdullah

```

Notice that a blank line is displayed after each line in the output.
This is because each item that is read from the file ends with a newline character (\n).
Later, you will learn how to remove the newline character.

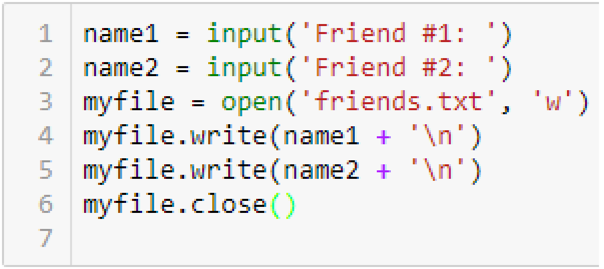
Dealing with the newline \n

Concatenating a Newline to a String

Earlier, we have written 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.

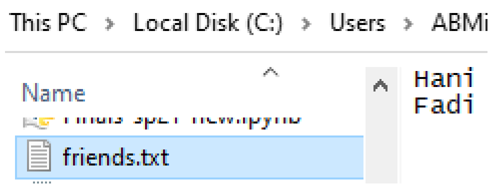
The following code demonstrates how this is done.

```
name1 = input('Friend #1: ‘)
name2 = input('Friend #2: ')
myfile = open('friends.txt',
'w’)
myfile.write(name1 + '\n’)
myfile.write(name2 + '\n')
myfile.close()
```



```
name1 = input('Friend #1: ')
name2 = input('Friend #2: ')
myfile = open('friends.txt', 'w')
myfile.write(name1 + '\n')
myfile.write(name2 + '\n')
myfile.close()
```

```
Friend #1: Hani
Friend #2: Fadi
```



This PC › Local Disk (C:) › Users › ABMi

Name

friends.txt

```
Hani
Fadi
```

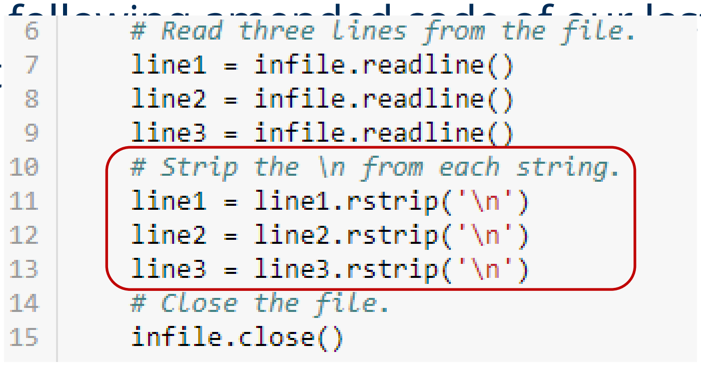
Dealing with the newline \n

Reading a String and Stripping the Newline from it

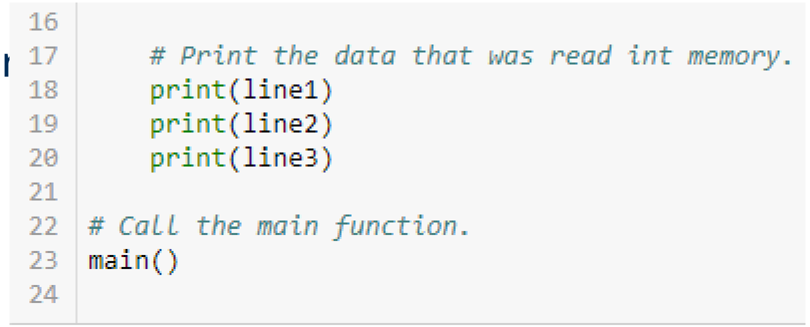
The \n serves a necessary purpose inside a file: it separates the items that are stored in the file.
However, in many cases, you want to remove the \n from a string after it is read from a file.

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 amended code of our last pr[illegible]
met[illegible]



```python
    # Read three lines from the file.
    line1 = infile.readline()
    line2 = infile.readline()
    line3 = infile.readline()
    # Strip the \n from each string.
    line1 = line1.rstrip('\n')
    line2 = line2.rstrip('\n')
    line3 = line3.rstrip('\n')
    # Close the file.
    infile.close()
```



```python

    # Print the data that was read int memory.
    print(line1)
    print(line2)
    print(line3)

# Call the main function.
main()

```

```
Ameen Sinjer
Mazen Ali
Nawaf Abdullah
```

As a result, the extra blank lines do not appear in the output.

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.

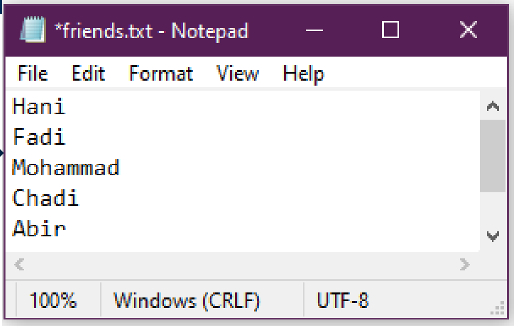
In Python, you can use the **'a'** mode to open an output file in ***append mode***, which means
the following:

- 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, consider our file friends.txt that contains the names **Hani** and **Fadi**, each in a separate line. The following code opens the file and appends additional data to its existing contents.

```python
myfile = open('friends.txt', 'a')
myfile.write('Mohammad\n')
myfile.write('Chadi\n')
myfile.write('Abir\n')
myfile.close()
```

After running this program, the file **friends.txt** will contain the following data



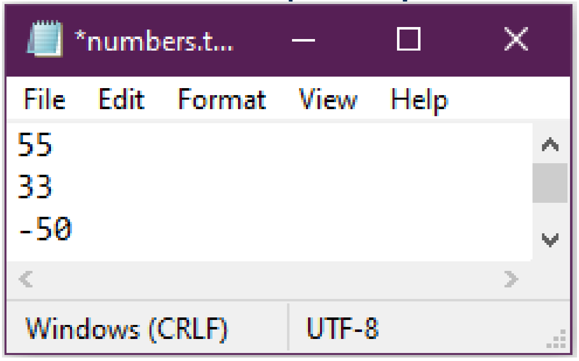
```
Hani
Fadi
Mohammad
Chadi
Abir
```

Writing and Reading Numeric Data

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'.
The below Program 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.

```
outfile = open('numbers.txt', 'w')
# Get three numbers from the user.
num1 = int(input('Enter a number: ')) #55
num2 = int(input('Enter another number: '))
# 33
num3 = int(input('Enter another number: '))
# -50
# Write the numbers to the file.
outfile.write(str(num1) + '\n')
outfile.write(str(num2) + '\n')
outfile.write(str(num3) + '\n')
outfile.close()
```

The expression **str(num1) + '\n'** converts the value referenced by num1 to a string and concatenates the \n escape



```
55
33
-50
```

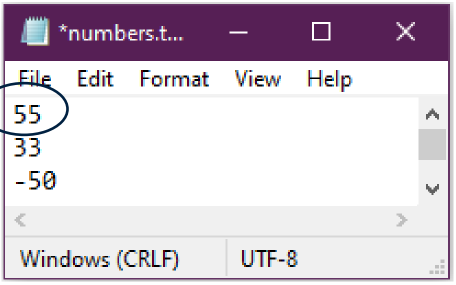
Writing and Reading Numeric Data

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-ca3d7435d60d> in <module>()
      2 value = infile.readline()
      3 infile.close()
----> 4 print(value+3)

TypeError: must be str, not int
```

The **value** variable referenced the string '55\n'. This caused a problem as we intend to perform math with the value variable, because we cannot perform math on strings.



```
55
33
-50
```

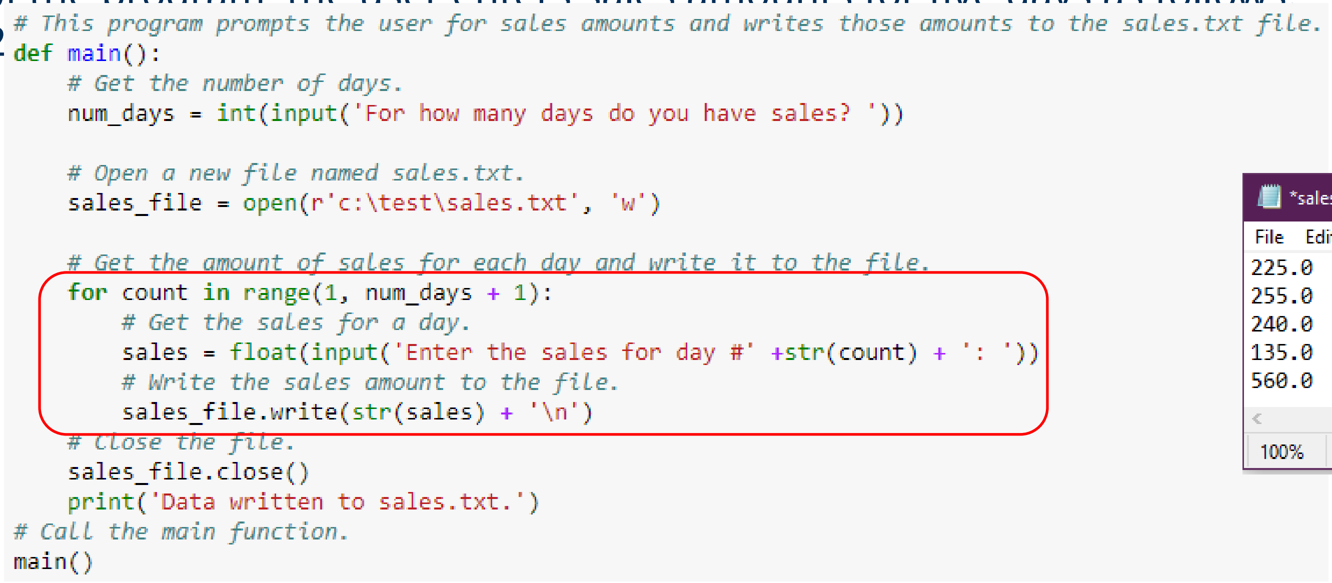
```
infile = open('numbers.txt', 'r')
value = int(infile.readline())
infile.close()
print(value+3)
```

```
58
```

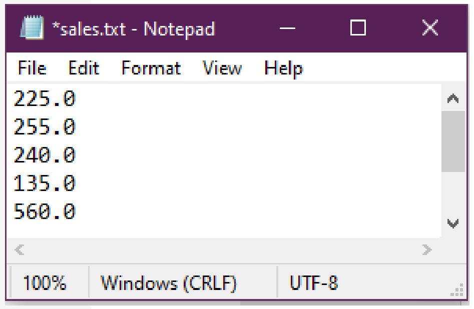
Notice in line 2 a call to the *readline* method is used as the argument to the *int* function. After this statement executes, the **value** variable will reference the integer 55, hence the displayed number will be 58.

Using loops to Process Files

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, consider the below program that 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 as follows:
225,2



```python
# This program prompts the user for sales amounts and writes those amounts to the sales.txt file.
def main():
    # Get the number of days.
    num_days = int(input('For how many days do you have sales? '))

    # Open a new file named sales.txt.
    sales_file = open(r'c:\test\sales.txt', 'w')

    # Get the amount of sales for each day and write it to the file.
    for count in range(1, num_days + 1):
        # Get the sales for a day.
        sales = float(input('Enter the sales for day #' +str(count) + ': '))
        # Write the sales amount to the file.
        sales_file.write(str(sales) + '\n')
    # Close the file.
    sales_file.close()
    print('Data written to sales.txt.')
# Call the main function.
main()
```



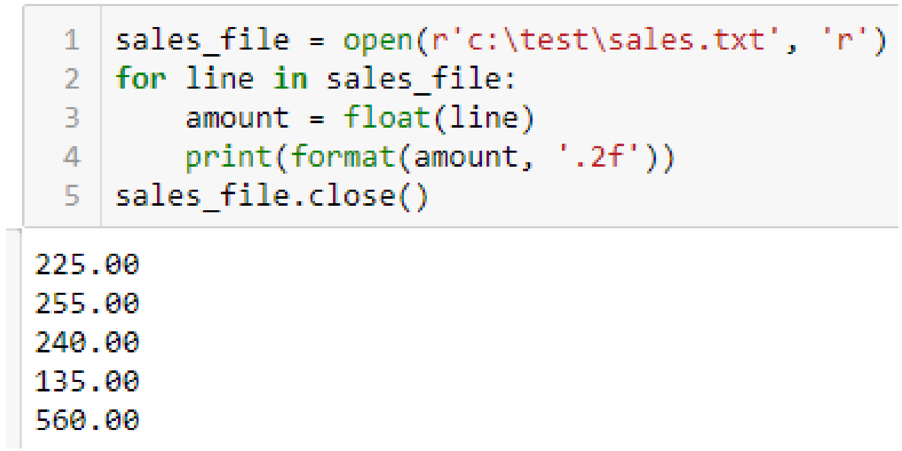
```
225.0
255.0
240.0
135.0
560.0
```

Using loops to Process Files

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

The Python language 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 automatically stops when the end of the file has been reached.

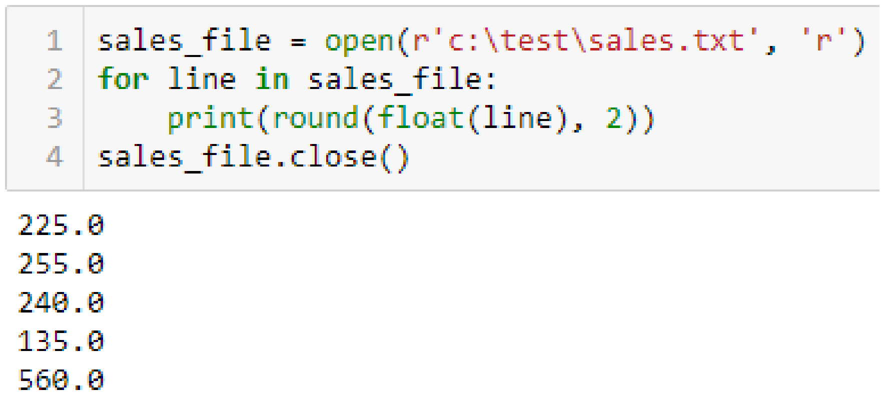
The below programs read and display all the items in the **sales.txt** file.



```
sales_file = open(r'c:\test\sales.txt', 'r')
for line in sales_file:
    amount = float(line)
    print(format(amount, '.2f'))
sales_file.close()
```

```
225.00
255.00
240.00
135.00
560.00
```

OR



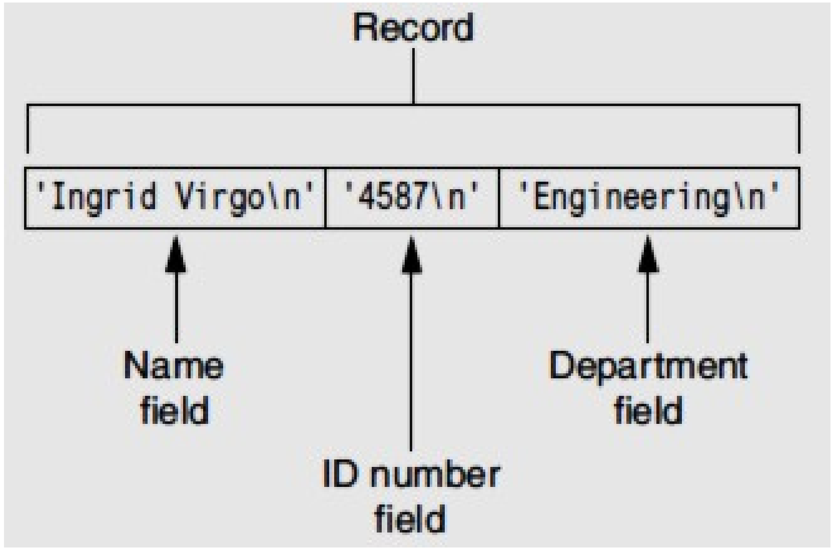
```
sales_file = open(r'c:\test\sales.txt', 'r')
for line in sales_file:
    print(round(float(line), 2))
sales_file.close()
```

```
225.0
255.0
240.0
135.0
560.0
```

Processing Records

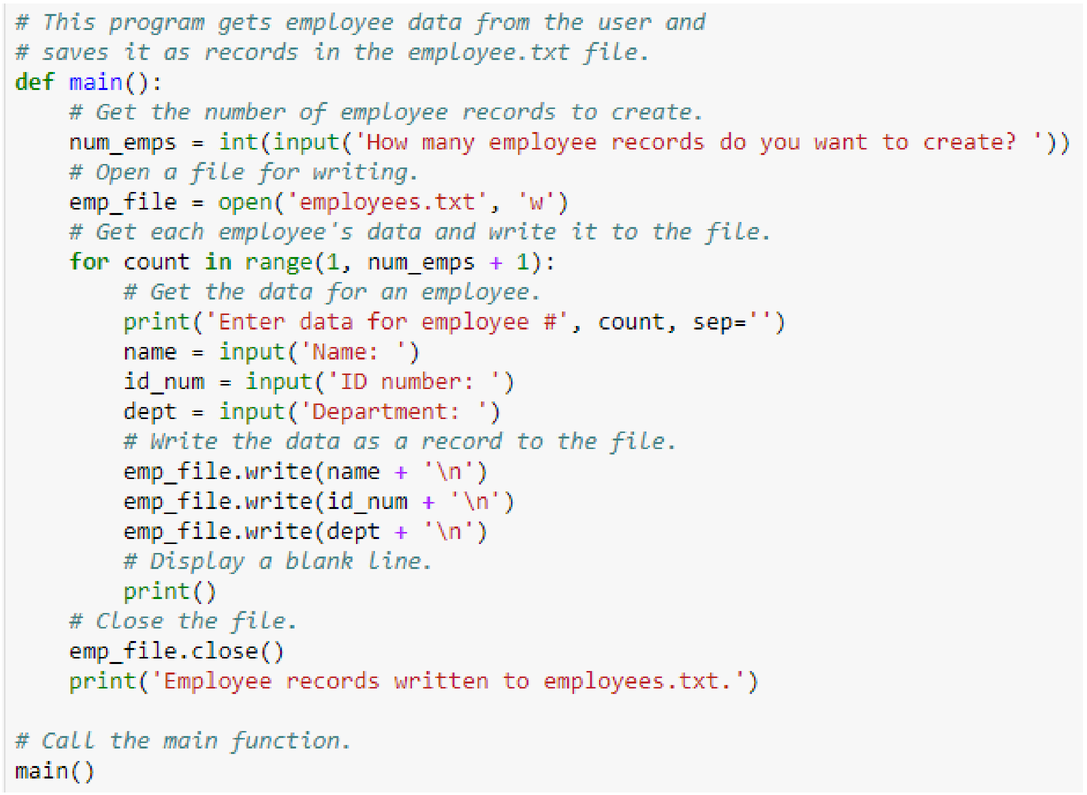
When data is written to a file, it is often organized into records and fields. A *record* is a complete set of data that describes one item, and a *field* is a single piece of data within a record.

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.
This is illustrated in the below figure.

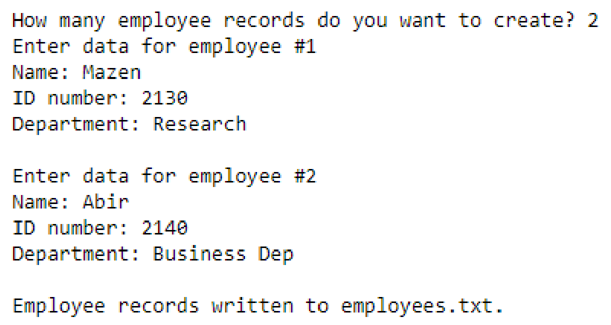


Processing Records

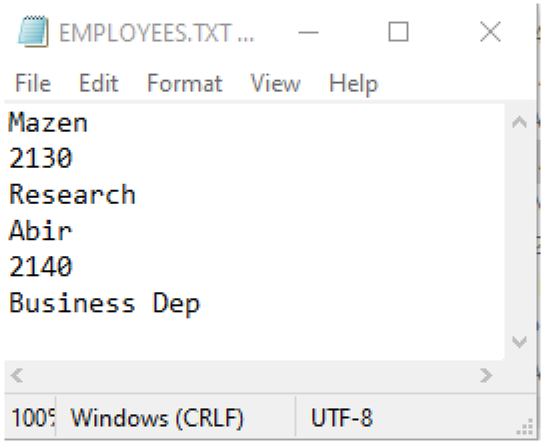
Each time you write a record to a sequential access file, you write the fields that make up the record, one after the other. For example, in a file that contains three employee records, each record will consist of the employee’s name, ID number, and department. The program below shows how a specific number of records can be written to a file.



```python
# This program gets employee data from the user and
# saves it as records in the employee.txt file.
def main():
    # Get the number of employee records to create.
    num_emps = int(input('How many employee records do you want to create? '))
    # Open a file for writing.
    emp_file = open('employees.txt', 'w')
    # Get each employee's data and write it to the file.
    for count in range(1, num_emps + 1):
        # Get the data for an employee.
        print('Enter data for employee #', count, sep='')
        name = input('Name: ')
        id_num = input('ID number: ')
        dept = input('Department: ')
        # Write the data as a record to the file.
        emp_file.write(name + '\n')
        emp_file.write(id_num + '\n')
        emp_file.write(dept + '\n')
        # Display a blank line.
        print()
    # Close the file.
    emp_file.close()
    print('Employee records written to employees.txt.')

# Call the main function.
main()
```



```
How many employee records do you want to create? 2
Enter data for employee #1
Name: Mazen
ID number: 2130
Department: Research

Enter data for employee #2
Name: Abir
ID number: 2140
Department: Business Dep

Employee records written to employees.txt.
```

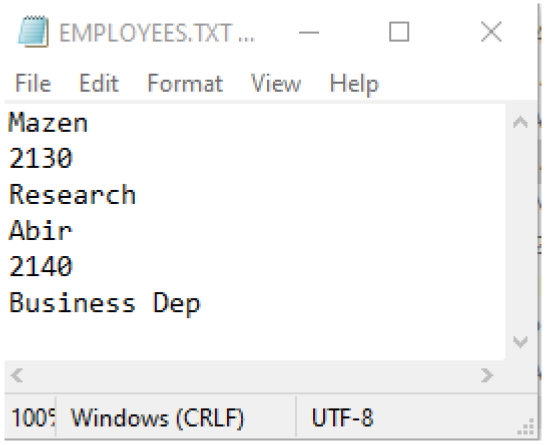


```
Mazen
2130
Research
Abir
2140
Business Dep
```

Processing Records

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. The below program demonstrates how we can read the employee records in the **employee.txt** file.

```python
# This program displays the records that are in the employees.txt file.
def main():
    # Open the employees.txt file.
    emp_file = open('employees.txt', 'r')
    # Read the first line from the file, which is the name field of the first record.
    name = emp_file.readline()
    # If a field was read, continue processing.
    while name != '':
        # Read the ID number field and the department field.
        id_num = emp_file.readline()
        dept = emp_file.readline()
        # Strip the newlines from the fields.
        name = name.rstrip('\n')
        id_num = id_num.rstrip('\n')
        dept = dept.rstrip('\n')
        # Display the record
        print('Name:', name)
        print('ID:', id_num)
        print('Dept:', dept)
        print()
        # Read the name field of the next record.
        name = emp_file.readline()
    # Close the file.
    emp_file.close()
# Call the main function.
main()
```

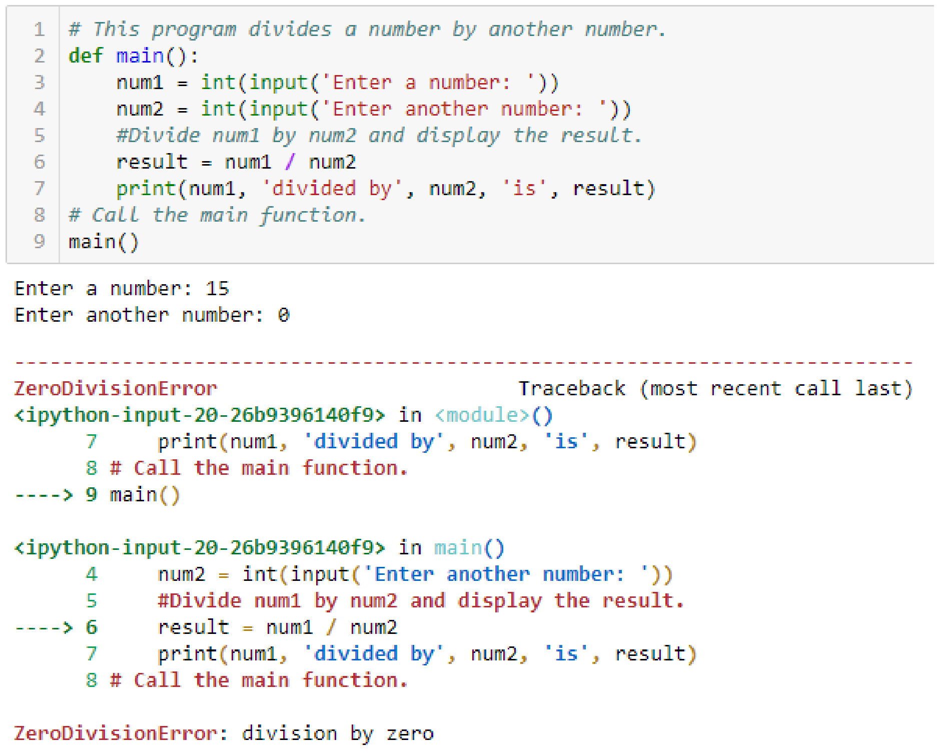


```
Mazen
2130
Research
Abir
2140
Business Dep
```

```
Name: Mazen
ID: 2130
Dept: Research

Name: Abir
ID: 2140
Dept: Business Dep
```

Exceptions

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



```
# This program divides a number by another number.
def main():
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    #Divide num1 by num2 and display the result.
    result = num1 / num2
    print(num1, 'divided by', num2, 'is', result)
# Call the main function.
main()
```

```
Enter a number: 15
Enter another number: 0

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-20-26b9396140f9> in <module>()
      7     print(num1, 'divided by', num2, 'is', result)
      8 # Call the main function.
----> 9 main()

<ipython-input-20-26b9396140f9> in main()
      4     num2 = int(input('Enter another number: '))
      5     #Divide num1 by num2 and display the result.
----> 6     result = num1 / num2
      7     print(num1, 'divided by', num2, 'is', result)
      8 # Call the main function.

ZeroDivisionError: division by zero
```

In the sample running of the program, an exception occurred because the user entered **zero** as the second number. (Division by 0 causes an exception because it is mathematically impossible.)

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)

Exceptions

You can prevent many exceptions from being raised by carefully coding your program. For example, 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.
Some exceptions, however, cannot be avoided regardless of how carefully you write your program.

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.
```

Exceptions

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

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***.

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.

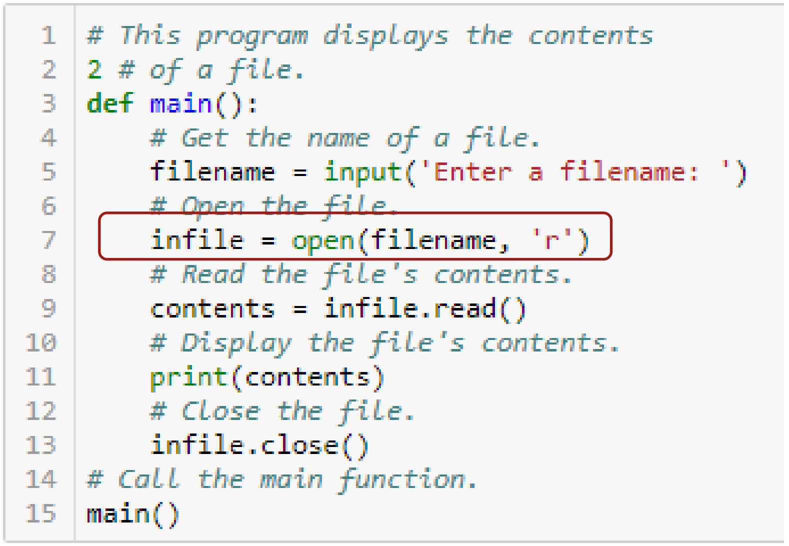
• If a statement in the *try suite* raises an exception that is *not* specified by the *ExceptionName* in an except clause, then the program will halt with a traceback error message.

• 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.

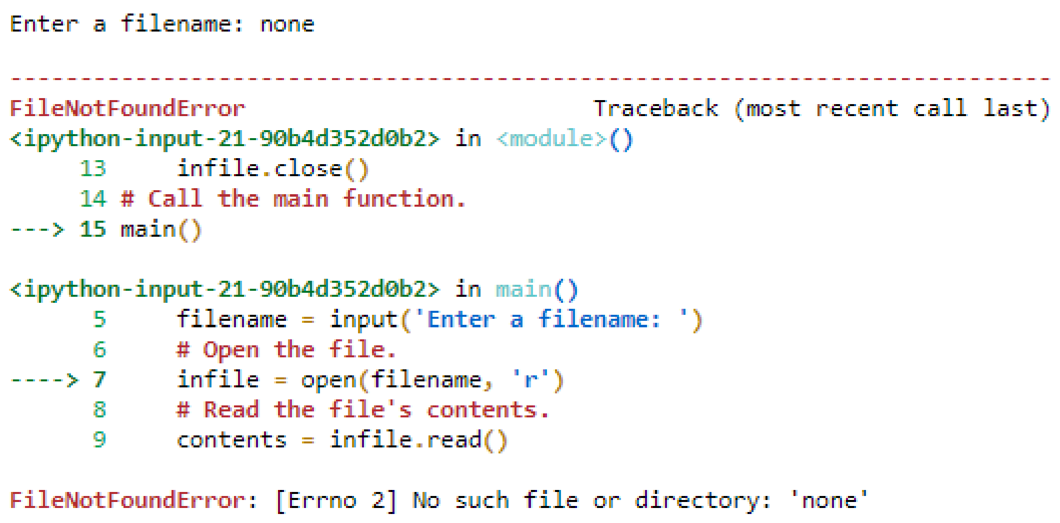
Exceptions

Let's look at an example. The below 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. This is what happened in the sample run



```
 1  # This program displays the contents
 2  2 # of a file.
 3  def main():
 4      # Get the name of a file.
 5      filename = input('Enter a filename: ')
 6      # Open the file.
 7      infile = open(filename, 'r')
 8      # Read the file's contents.
 9      contents = infile.read()
10      # Display the file's contents.
11      print(contents)
12      # Close the file.
13      infile.close()
14  # Call the main function.
15  main()
```

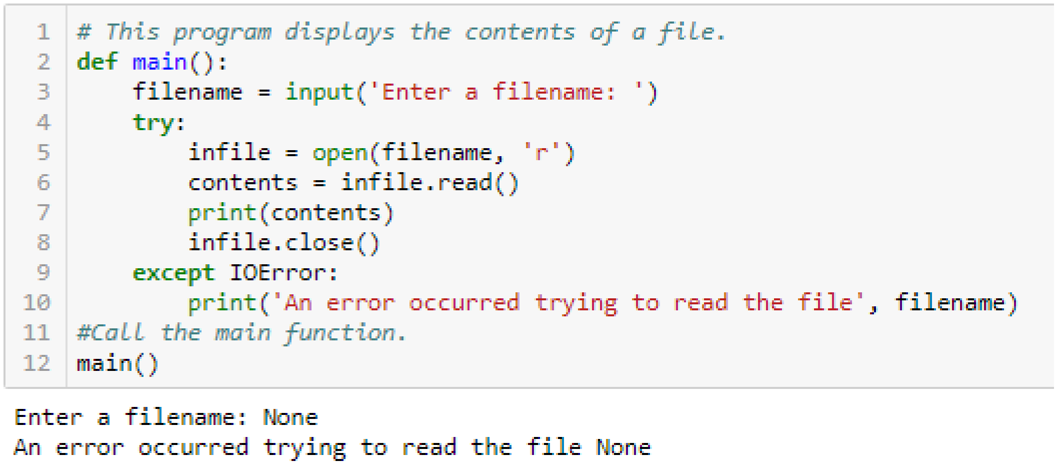


```
Enter a filename: none

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-21-90b4d352d0b2> in <module>()
     13     infile.close()
     14 # Call the main function.
---> 15 main()

<ipython-input-21-90b4d352d0b2> in main()
      5     filename = input('Enter a filename: ')
      6     # Open the file.
----> 7     infile = open(filename, 'r')
      8     # Read the file's contents.
      9     contents = infile.read()

FileNotFoundError: [Errno 2] No such file or directory: 'none'
```

The statement in line 7 raised the exception when it called the open function. 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.

Exceptions

The below program shows how we can modify the previous program with a try/except statement that gracefully responds to an ***IOError*** exception.



```
 1  # This program displays the contents of a file.
 2  def main():
 3      filename = input('Enter a filename: ')
 4      try:
 5          infile = open(filename, 'r')
 6          contents = infile.read()
 7          print(contents)
 8          infile.close()
 9      except IOError:
10          print('An error occurred trying to read the file', filename)
11  #Call the main function.
12  main()
```

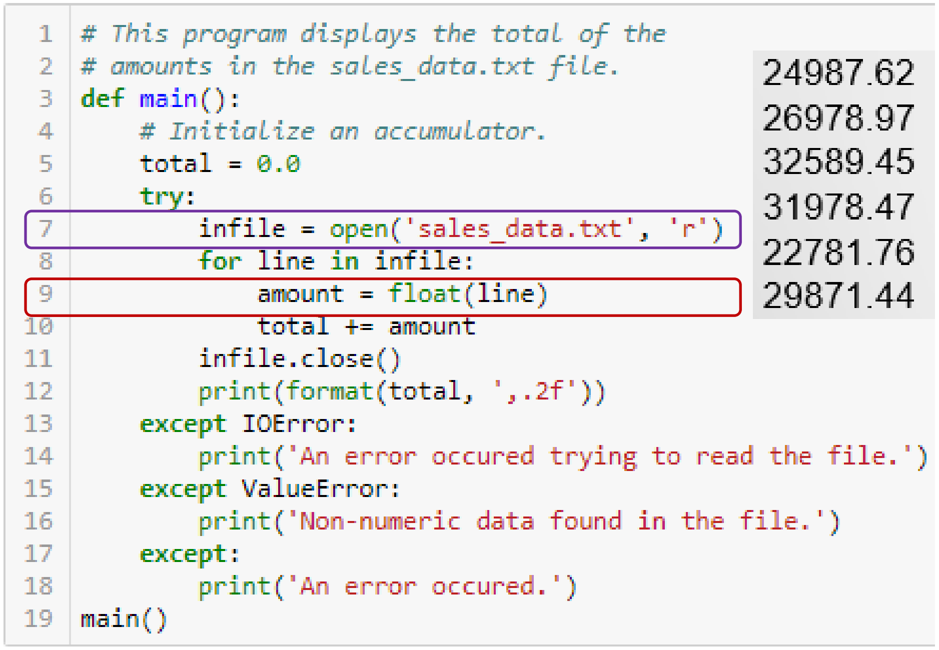
```
Enter a filename: None
An error occurred trying to read the file None
```

Let's look at what happened in the sample run. When line 3 executed, the user entered None, which was assigned to the filename variable. Inside the ***try suite***, line 5 attempts to open the file None. Because this file does not exist, the statement raises an IOError exception. When this happens, the program exits the try suite, skipping lines 6 through 8. Because the except clause in line 9 specifies the IOError exception, the program jumps to the handler that begins in line 10.

Handling Multiple Exceptions

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. For example, our next program reads the contents of a file named sales_data.txt.

Each line in the file contains the sales amount for one month, and the file has several



```
 1  # This program displays the total of the
 2  # amounts in the sales_data.txt file.
 3  def main():
 4      # Initialize an accumulator.
 5      total = 0.0
 6      try:
 7          infile = open('sales_data.txt', 'r')
 8          for line in infile:
 9              amount = float(line)
10              total += amount
11          infile.close()
12          print(format(total, ',.2f'))
13      except IOError:
14          print('An error occured trying to read the file.')
15      except ValueError:
16          print('Non-numeric data found in the file.')
17      except:
18          print('An error occured.')
19  main()
```

```
24987.62
26978.97
32589.45
31978.47
22781.76
29871.44
```

- The statement in line 7 can raise an IOError exception if the sales_data.txt file does not exist. The for loop in line 8 can also raise an IOError exception if it encounters a problem reading data from the file.
- The float function in line 9 can raise a ValueError exception if the line variable references a string that cannot be converted to a floating-point number (an alphabetic string, for example).
- The except clause in line 13 specifies the IOError exception. Its handler in line 14 will execute if an IOError exception is raised.
- The except clause in line 15 specifies the ValueError exception. Its handler in line 16 will execute if a ValueError exception is raised.
- The except clause in line 17 does not list a specific exception. Its handler in line 18 will execute if an exception that is not handled by the other except clauses is raised.

Handling Multiple Exceptions

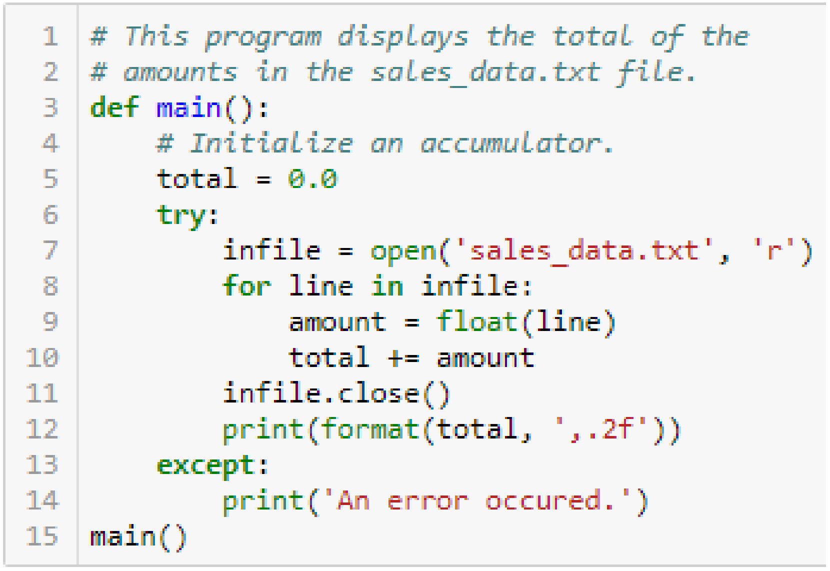
If an exception occurs in the try suite, the Python interpreter examines each of the except clauses, from top to bottom, in the try/except statement.
When it finds an except clause that specifies a type that matches the type of exception that occurred, it branches to that except clause. If none of the except clauses specifies a type that matches the exception, the interpreter branches to the except clause in line 17.

Using One except Clause to Catch All Exceptions

The previous example demonstrated how multiple types of exceptions can be handled individually in a try/except statement.

Sometimes you might want to write a try/except statement that simply catches any exception that is raised in the try suite and, regardless of the exception's type, responds the same way.

You can accomplish that in a try/except statement by writing one except clause that does not specify a particular type of exception.



```python
# This program displays the total of the
# amounts in the sales_data.txt file.
def main():
    # Initialize an accumulator.
    total = 0.0
    try:
        infile = open('sales_data.txt', 'r')
        for line in infile:
            amount = float(line)
            total += amount
        infile.close()
        print(format(total, ',.2f'))
    except:
        print('An error occured.')
main()
```

```
An error occured.
```

What If an Exception Is Not Handled?

Unless an exception is handled, it will cause the program to halt. There are two possible ways for a thrown exception to go unhandled. The first possibility is for the try/except statement to contain no except clauses specifying an exception of the right type.
The second possibility is for the exception to be raised from outside a try suite.
In either case, the exception will cause the program to halt.

In this section, you’ve seen examples of programs that can raise ZeroDivisionError exceptions, IOError exceptions, and ValueError exceptions.
There are many different types of exceptions that can occur in a Python program. When you are designing try/except statements, one way you can learn about the exceptions that you need to handle is to consult the Python documentation. It gives detailed information about each possible exception and the types of errors that can cause them to occur.

Another way that you can learn about the exceptions that can occur in a program is through experimentation.
You can run a program and deliberately perform actions that will cause errors. By watching the traceback error messages that are displayed, you will see the names of the exceptions that are raised. You can then write except clauses to handle these exceptions.

Exercise 1

1. Assume a file containing a series of integers is named numbers.txt and exists on the computer's disk.
   Write a program that calculates the average of all the numbers stored in the file.

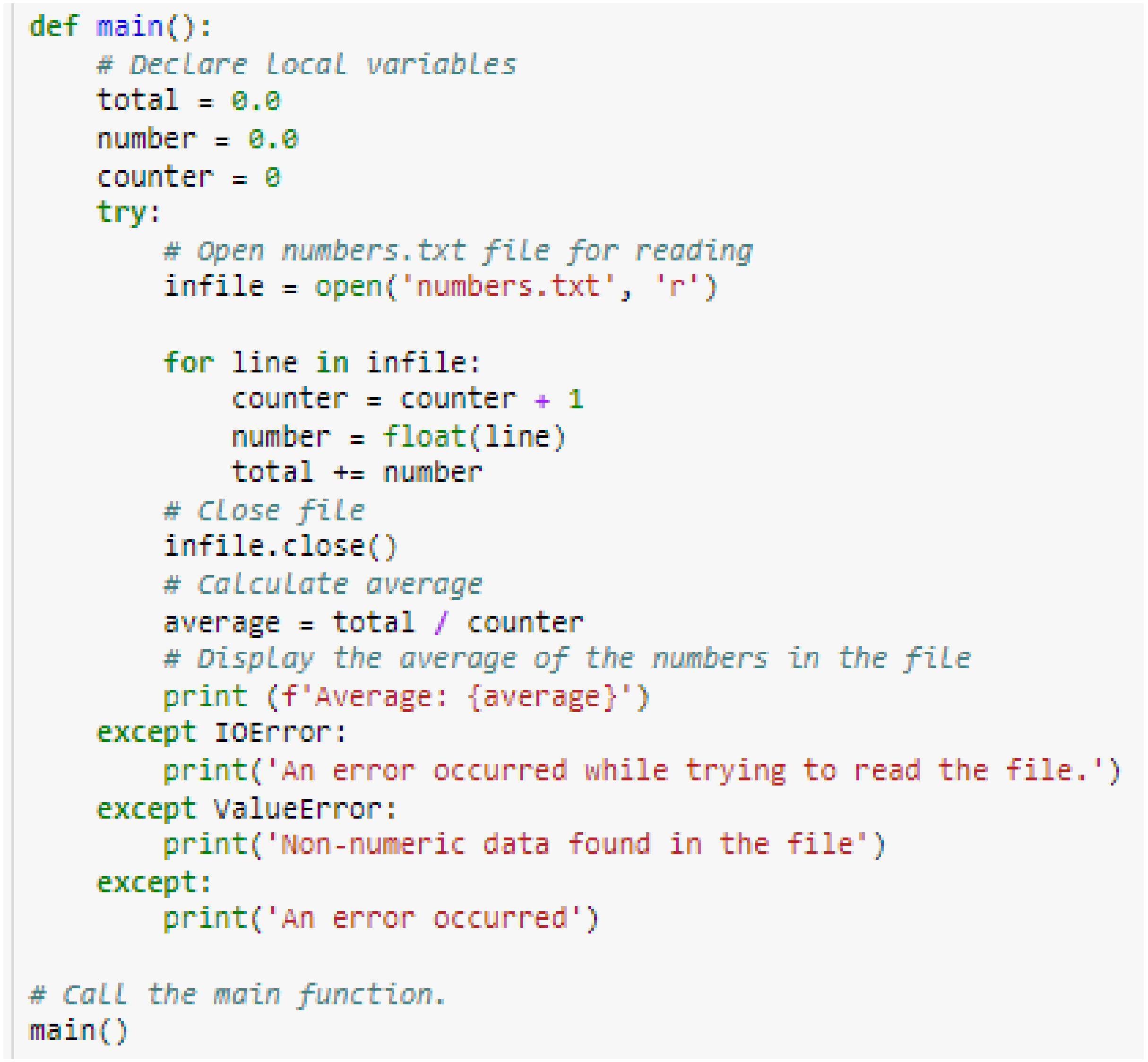
Exercise 1- Solution

```
def main():
    # Declare variables
    total = 0.0
    number = 0.0
    counter = 0
    # Open numbers.txt file for reading
    infile = open('numbers.txt', 'r')
    for line in infile:
        counter = counter + 1
        number = float(line)
        total += number
    # Close file
    infile.close()
   # Calculate average
    average = total / counter
    # Display the average of the numbers in the
file
    print(f'Average: {average}')

# Call the main function.
main()
```

Exercise 2

2. Modify the program that you wrote for Exercise 1, so it handles the following exceptions:
   - It should handle any *IOError* exceptions that are raised when the file is opened, and data is read from it.
   - It should handle any *ValueError* exceptions that are raised when the items that are read from the file are converted to a number.

Exercise 2- Solution



```python
def main():
    # Declare local variables
    total = 0.0
    number = 0.0
    counter = 0
    try:
        # Open numbers.txt file for reading
        infile = open('numbers.txt', 'r')

        for line in infile:
            counter = counter + 1
            number = float(line)
            total += number
        # Close file
        infile.close()
        # Calculate average
        average = total / counter
        # Display the average of the numbers in the file
        print (f'Average: {average}')
    except IOError:
        print('An error occurred while trying to read the file.')
    except ValueError:
        print('Non-numeric data found in the file')
    except:
        print('An error occurred')

# Call the main function.
main()
```