Computer Programming in Python

Chapter 7

File Operations and Dialogs

Chapter 7 File Operations

- Files
  - Store information and data used by computers
    - Data stored in RAM does not persist between runs of the program, or when the computer is turned off
    - Files allow information to be stored until it is needed, changed when required, and deleted when no longer needed

Chapter 7 File Operations

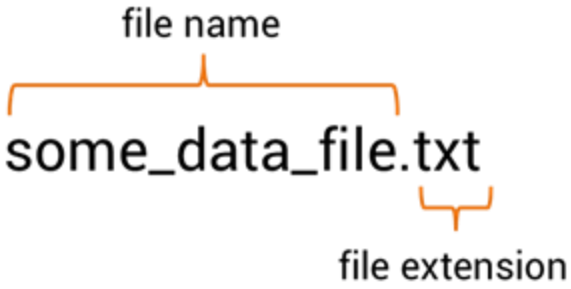
- File Names have a ***file extension***
  - Three or four letters that follow the period in the file name

    spreadsheet.xlsx

  - File extensions are used by most operating systems to associate the file with an application
    - When a file is double-clicked, the OS determines the application to launch based upon the file's extension and the application that was used to open that type of file previously

Chapter 7 File Operations

- File Extensions
  - Double-clicking a file named “song.mp3” will launch an audio player because the audio player application has been associated with the mp3 file extension
  - The “txt” file extension is typical for text files which are usually opened with Notepad or Notes by the computer’s operating system



Chapter 7 File Operations

- File Extensions
  - Applications

Extension	Description
.docx	Microsoft Word document file
.exe	executable file
.html	web page file
.java	Java source code file
.jpg	JPEG image file
.mov	movie file
.mp3	audio file
.pdf	Adobe Portable Document File
.py	Python source code file
.zip	ZIP compressed archive

Chapter 7 File Operations

- File Operations
  - Files being read from by a program are typically referred to as ***input files***
  - Files being written to as ***output files***
  - Three (3) steps to using a file in a computer program
    1. The file is opened
    2. The file is processed (either written to or read from)
    3. The file is closed

Chapter 7 File Operations

- Opening a File
  - When a file is opened using Python, it is associated with the program through a ***file object*** that has a variable reference
  - The variable reference is the name to be associated with the file in the program
    - This is not that different from the way that an integer or float is defined except that the name is associated with a file object
  - The general format for opening a file is:

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

Chapter 7 File Operations

- Opening a File
  - The ***open*** function is passed two arguments
    - The first is the actual name of the file
    - The second is the ***mode*** in which the file will be opened
      - Determines the way that the file will be opened, and what will occur if the file exists or if it does not

```
my_file = open('some_data.txt', 'r')
```

reference name

name of file

read mode

Chapter 7 File Operations

- Opening a File
  - When the file name is used as the first argument, the program will search the ***default directory*** for the file
    - Where the program is running
  - A full path to the file can also be used
    - Requires ‘r’ before the path to the file

```
myFile = open(r 'C:\Users\csimber\some_data.txt', 'r')
```

tells Python to disregard the backslashes in the path

path to file

Chapter 7 File Operations

- Opening a File
  - The ***mode*** determines the file operation

Mode	Description
'r'	Opens a file for reading, produces an error if the file does not exist
'w'	Opens a file for writing. If the file exists, the contents is erased. If the file does not exist, it creates the file.
'a'	Opens a file for appending. Creates the file if it does not exist.

Chapter 7 File Operations

- Opening a File
  - File objects have methods that simplify some file handling processes
  - Once a file object is associated with a variable, the variable name is used to access the methods
  - The only time that the actual file name is used is when the file is being opened

```
my_file = open('some_data.txt', 'r')
```

variable reference

open in read mode

Chapter 7 File Operations

- Writing to a File
  - When writing to a file, the ***write()*** method is used and is passed what is to be written
  - The variable reference assigned to the file is followed by the dot operator, and the method name

```
my_file = open('data_file.txt', 'w')

my_file.write('A stitch in time saves nine.')

my_file.close()
```

Chapter 7 File Operations

- Closing Files
  - Using the ***close()*** method ensures that no data is lost
  - Data being written to a file is queued in a ***buffer*** (a holding area in memory) for efficiency
  - Closing the file in the program forces anything being held in the buffer to be written to the file before it is closed
  - If a program does not close the file, the operating system will eventually close it, but would not check the buffer first

Chapter 7 File Operations

- Writing to a File
  - The *write()* method will do as it is told, and if the data is to be written on separate lines, then line feeds need to be incorporated into the write statement
    - This is unlike print which automatically adds a line feed
  - The escape sequence ‘\n’ is the newline character and is used to produce a line feed in the file

Chapter 7 File Operations

- Writing to a File
  - This example opens a file named “test_file.txt” for writing, associates it with out_file, writes three phrases on separate lines in the file, and closes the file

Line feed

```
def main():
    out_file = open('test_file.txt', 'w')

    out_file.write('Line #1: The first line.\n')
    out_file.write('Line #2: The second line.\n')
    out_file.write('Line #3: The third line.\n')
    out_file.write('Line #4: The fourth line.')

    out_file.close()

main( )
```

Chapter 7 File Operations

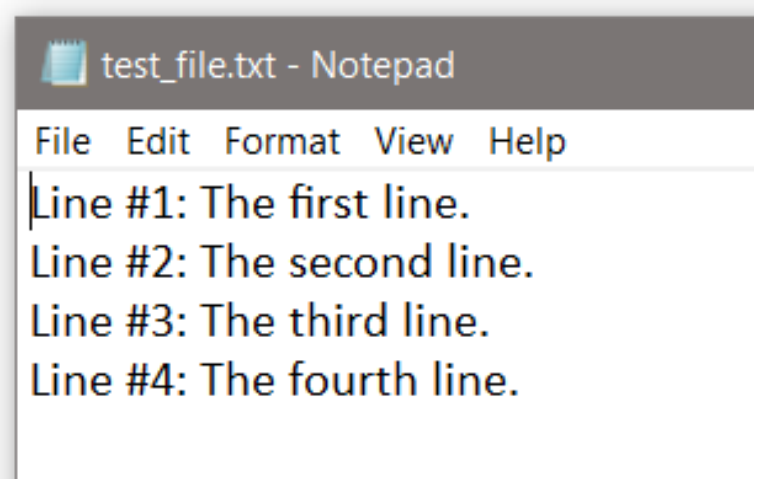
- Writing to a File
  - The example program created the new file, opened it, wrote the lines, and closed the file

```
def main():
    out_file = open('test_fi

    out_file.write('Line #1:
    out_file.write('Line #2:
    out_file.write('Line #3:
    out_file.write('Line #4:

    out_file.close()

main( )
```



Chapter 7 File Operations

- Writing to a File
  - Writing the contents of a variable to a file is handled much like the print function
  - For a line feed, the newline character is concatenated onto a string variable

```
out_file.write(my_string + '\n')
```

- Writing to a File
  - If the value to be written is not a string, the ***str*** function must be used to convert it to a string
  - Numeric values cannot be written to files as numeric values in Python and must be converted to strings

```
out_file.write(str(my_int) + '\n')
```

Trying to write a numeric value will cause a TypeError

Chapter 7 File Operations

- Writing to a File
  - Converting values to strings example

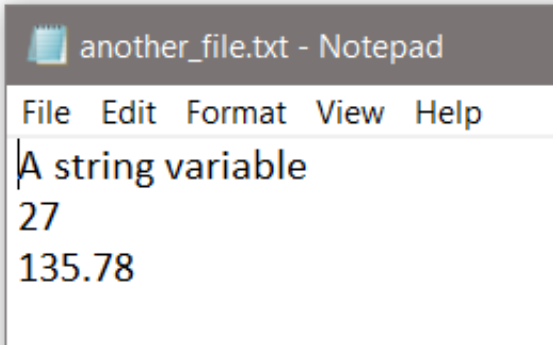
```
def main():

    my_string = 'A string variable'
    my_int = 27
    my_float = 135.78

    out_file = open('another_file.txt', 'w')
    out_file.write(my_string + '\n')
    out_file.write(str(my_int) + '\n')
    out_file.write(str(my_float))

    out_file.close()

main( )
```



Chapter 7 File Operations

- Writing to a File - Appending
  - Opening an existing file in write mode erases any data that had been stored in the file
    - What actually takes place is that the old file is deleted, and a new empty file is created
  - To append data to existing data, the file is opened in append mode using ‘a’
    - Any existing data in the file is preserved

Chapter 7 File Operations

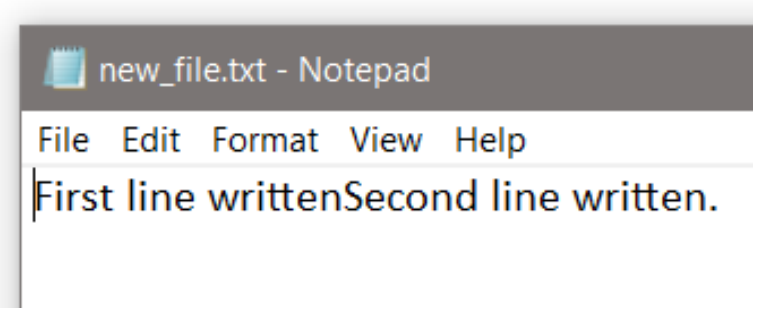
- Writing to a File - Appending
  - Opening an existing file in append mode

```
out_file = open('new_file.txt', 'w')
out_file.write('First line written')
out_file.close()

out_file = open('new_file.txt', 'a')
out_file.write('Second line written.')
out_file.close()
```

append mode

No line feed was added



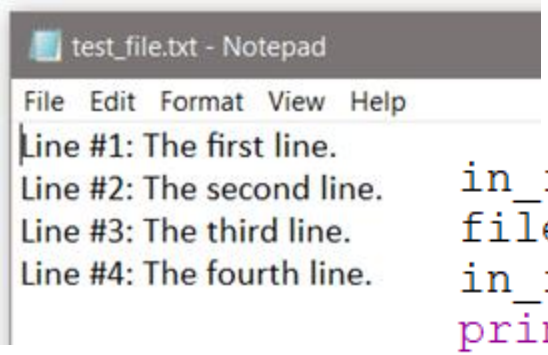
Chapter 7 File Operations

- Reading from a File
  - Open the file using ‘r’ as the mode to read
  - File object methods for reading
    - ***read()*** which returns the entire file contents as a string
    - ***readline()*** which will read one line from the file (until ‘\n’ is encountered)

Since the read() method also reads the newline characters, the information read will include any line feeds

Chapter 7 File Operations

- Reading from a File
  - The example reads the entire file (including line feeds) into the variable *file_data*, closes the file, and prints the variable (which includes line feeds)

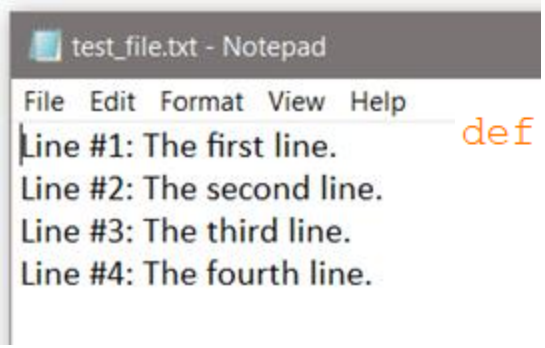



```
in_file = open('test_file.txt', 'r')
file_data = in_file.read()
in_file.close()
print(file_data)
```

```
Line #1: The first line.
Line #2: The second line.
Line #3: The third line.
Line #4: The fourth line.
```

Chapter 7 File Operations

- Reading from a File
  - The example reads a single line from the file into the variable *file_data,* closes the file, and prints the variable




```
def main():
    in_file = open('test_file.txt', 'r')

    one_line = in_file.readline()

    in_file.close()

    print(one_line)

main( )
```

```
Line #1: The first line.
```

Chapter 7 File Operations

- Reading Numeric Data from a File
  - When reading numeric values from a file, they are returned as strings
  - Must be converted to a numeric data type in order to use them as a numeric value
  - Chapter 3 introduced casting for type conversion which is used when reading from a file

The data format in the file may cause issues when casting

Chapter 7 File Operations

- Reading Numeric Data from a File
  - Since Readline reads until the line feed, there is no issue here

numbers.txt - Notepad

File Edit Format View Help

```
27
14
123.45
```

```
in_file = open('numbers.txt', 'r')

num1 = int(in_file.readline())
num2 = int(in_file.readline())
num3 = float(in_file.readline())

num4 = num1 + num2 + num3
print(num4)
```

```
164.45
>>>
```

Chapter 7 File Operations

- Reading Data from a File
  - Typically, a loop is used when handling file data
  - One option is to read a line or value, process the data, and output some result
    - The loop continues to read until there are no more values
    - Every file contains an end of file (EOF) marker that indicates where the file ends
    - When it is reached, a value cannot be read by the Python method being used
      - This ends the loop that is reading from the file

Chapter 7 File Operations

- Reading Data from a File
  - Another option is to read the entire file into a string, and use the loop to parse the string
    - The loop continues to process until there are no more values
    - The parsing algorithm is dependent upon the data format

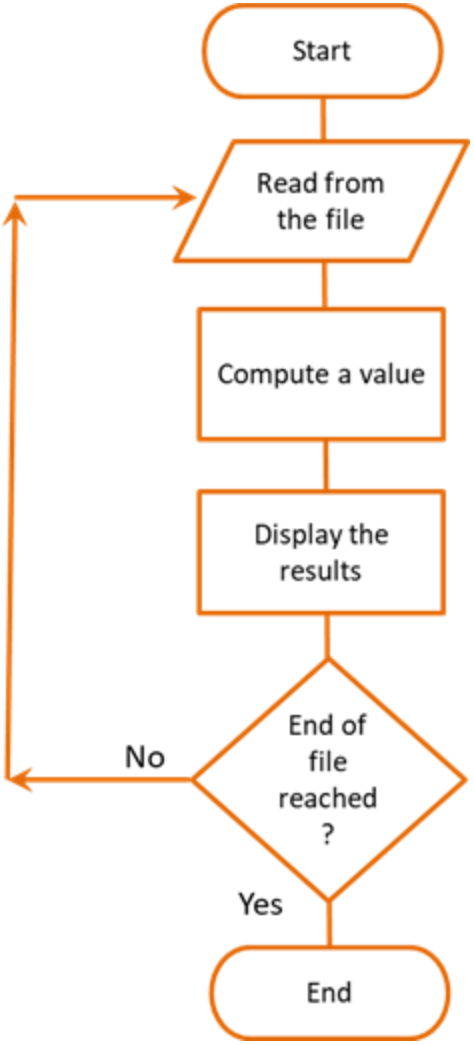
The file data format affects the parsing algorithm

Chapter 7 File Operations

- Reading from a File
  - File Reading Code and Flowchart

```
input_file = open('dataFile.txt', 'r')

for line in input_file:
    print(line)
```



Chapter 7 File Operations

- Reading Data from a File
  - The algorithm used to read and process file data is dependent to a large degree on the format of the data in the file
  - Therefore, how the data will be read and processed is a consideration when designing the writing format
    - One value per line
    - Columnar data with tabs between values
    - Values separated by a character or space

Chapter 7 File Operations

- Reading Delimited Data from a File
  - A ***delimiter*** is a character used to mark the beginning or end of an item of data
  - Consider a file written with columnar data with tabs between the values (tab-delimited data)
  - Using *read()* or *readline()* would include the delimiter (tabs in this case) in the returned string
  - It is common to read files one line at a time in a loop and process the data
  - Python has several methods that help to convert the data to a useful format

Chapter 7 File Operations

- Reading Files - Removing Characters
  - When Python reads from a file, the data is returned as a string and may include tabs, line feeds, and spaces
  - To remove tabs, line feeds, and spaces, there are several string modification methods including:
    - ***rstrip*** - removes white space (\n, \t, and space) from the right side of the string
    - ***lstrip*** - removes white space (\n, \t, and space) from the leading side of the string

Chapter 7 File Operations

- String Modification Methods

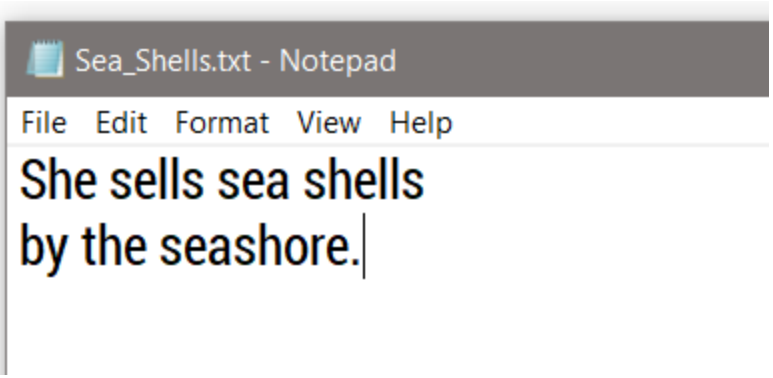
Method	Description
lower()	returns a lower case copy of the string
lstrip()	returns a copy of the string with leading white space removed
lstrip(*char*)	returns a copy of the string with leading instances of *char* removed
rstrip()	returns a copy of the string with trailing white space removed
rstrip(*char*)	returns a copy of the string with trailing instances of *char* removed
strip()	returns a copy of the string with all leading and trailing white space characters removed
strip(*char*)	returns a copy of the string with all leading and trailing instances of *char* removed
upper()	returns an upper case copy of the string

Chapter 7 File Operations

- String Modification Methods
  - Used to convert what has been read into a usable format and ensure that white space characters are not part of any data being converted to a numeric value
  - There is also a ***split()*** method that can split (parse) a line of data using a delimiter
    - The default delimiter for split() is any white space, but another character can be used

Chapter 7 File Operations

- Split example
  - A data file contains the phrase “She sells sea shells by the seashore” on two lines
  - The entire file will be read into a string
  - The *split()* method will extract each word because the default split character is whitespace (tabs, line feeds, spaces)



Chapter 7 File Operations

- Split example
  - The entire file is read into the variable phrase
  - The *split()* method extracts each word in a loop

```
def main():

    inFile = open('Sea_shells.txt', 'r')
    phrase = inFile.read()

    for word in phrase.split():
        print(word)

main( )
```

```
She
sells
sea
shells
by
the
seashore
```

Chapter 7 File Operations

- Split Example – Numeric Data
  - This example reads the entire file into ‘numbers’, and then uses split to separate them

numbers2.txt - Notepad

File Edit Format View Help

```
17
22
35
3.67
```

```
def main():

    total = 0.0

    inFile = open('numbers2.txt', 'r')
    numbers = inFile.read()

    for num in numbers.split():
        total = total + float(num)
    print(total)

main( )
```

```
77.67
>>>
```

Chapter 7 File Operations

- Split Example – Numeric Data
  - Notice that the file can be read directly one item at a time using a for-in loop (the line feeds are not an issue)

numbers2.txt - Notepad

File Edit Format View Help

```
17
22
35
3.67
```

```
total = 0.0

inFile = open('numbers2.txt', 'r')

for num in inFile:
    total = total + float(num)
print(total)
```

```
77.67
>>>
```

Chapter 7 File Operations

- Which technique to use
  - The technique used for reading and handling data from a file is often dependent upon the data format and the processing required
    - The data can be read one item or line at a time
    - The entire contents can be read at once
    - A loop can be used to read the data or to extract the individual values

The best technique may depend on the format of the file data

Chapter 7 File Operations

- Reading using a loop
  - A loop can read and process without storing the items in a separate variable before handling them

```
def main():

    inFile = open('numbers3.txt', 'r')

    for number in inFile:
        print(int(number) * 3.5)

    inFile.close()

main( )
```

Chapter 7 File Operations

- File handling issues
  - When a designing and developing a program that uses files, consider what happens if:
    - The file that the program reads from does not exist
    - The file that the program reads from cannot be opened
    - The file that the program reads from is corrupted
    - The file the program writes to cannot be created
      - The user does not have permission to create files
      - There is not enough room on the drive to create a file

Chapter 7 File Operations

- Exceptions
  - When a file cannot be created or cannot be opened, or when there is a data type mismatch, an exception will be raised (thrown) by the program
  - An ***exception*** is a type of error that occurs when a program is running
  - An exception must be *handled* or the program will terminate

Chapter 7 File Operations

- Exceptions
  - The format for an ***exception handler*** in Python is the ***try/except*** statement

statements that may raise an exception

```
try:
    statement1
    statement2
    etc.

except ExceptionName:
    statement1
    statement2
    etc.
```

statements to execute if an exception is raised

Chapter 7 File Operations

- Exceptions
  - The ***try*** block is entered and if a statement raises an exception, the handler immediately following the ***except*** clause that matches the type of exception raised executes and the program continues

```
try:
    statement1
    statement2
    etc.

except ExceptionName:
    statement1
    statement2
    etc.
```

Chapter 7 File Operations

- Exceptions – File Not Found
  - The code below could be stated:

    Try to open the file, and if an IOError occurs print “No file exists.”

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')
        print('If an exception is raised, ', end='')
        print('this line will not be displayed')

    except IOError:
        print('No file exists.')

    inFile.close()

main()
```

Chapter 7 File Operations

- Exceptions – File Not Found
  - The exception name is ***IOError*** which is the type of exception that would be raised if the file did not exist or could not be opened

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')
        print('If an exception is raised, ', end='')
        print('this line will not be displayed')

    except IOError:
        print('No file exists.')

    inFile.close()

main()
```

Chapter 7 File Operations

- Exceptions – File Not Found
  - Once an exception is raised, the try block is exited and any statements following the one that raised the exception will not be executed

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')
        print('If an exception is raised, ', end='')
        print('this line will not be displayed')

    except IOError:
        print('No file exists.')

    inFile.close()

main()
```

Chapter 7 File Operations

- Exceptions – Other Types
  - Each type of exception that could be raised should have an exception handler for that specific exception
    - An exception that is not handled will halt the program
  - An exception clause that does not list a specific exception, will handle any exception that is raised in the try suite
    - This could be considered a ***default handler***

An exception that is not handled will halt program execution

Chapter 7 File Operations

- Exceptions – Other Types
  - The two anticipated exceptions are a file error and a type error

```
try:
    input_file = open('missingFile.txt', 'r')
    for line in input_file:
        val = int(line)
        sum = sum + val

except IOError:
    print('No file exists.')

except ValueError:
    print('A bad value was read')

except:
    print('Other Error in program.')
```

default handler

Chapter 7 File Operations

- Exceptions
  - An exception raised is actually an object and contains information about the error
  - The contents is the same message that would be seen in the trace back error message
    - Can be accessed by assigning the exception to a variable

```
except ValueError as e:
        print(e)
```

Chapter 7 File Operations

- Exceptions
  - The try-except suite can include an ***else*** clause
    - Executes only if no exceptions were raised
    - If an exception is raised, then the else clause is skipped

“try to execute these, and if an exception is raised, execute the exception handler, *otherwise* execute these”.

Chapter 7 File Operations

- Exceptions
  - There is also an optional ***finally*** clause
    - Executes regardless of whether an exception was raised or not to perform cleanup
    - If a try suite opens a file and then executes other statements, one of those other statements may throw an exception
      - But the file is still open
    - A finally suite allows closing the file, or any other cleanup needed whether an exception was raised or not

Chapter 7 File Operations

- The ***finally*** clause (or finally suite)

```
def main():
    try:
        inFile = open('missingFile.txt', 'r')
        for line in input_file:
            val = int(line)

    except IOError:
        print('No File exists.')

    except ValueError:
        print('A bad value was read.')

    finally:
        inFile.close()

main()
```

Chapter 7 File Operations

- File Selection
  - Dialog boxes can simplify file handling when a user is selecting a file
  - Rather than have a user type a file name or path to a file, the dialog allows selection
  - This avoids typographical errors, and the dialog only displays files that exist

Chapter 7 File Operations

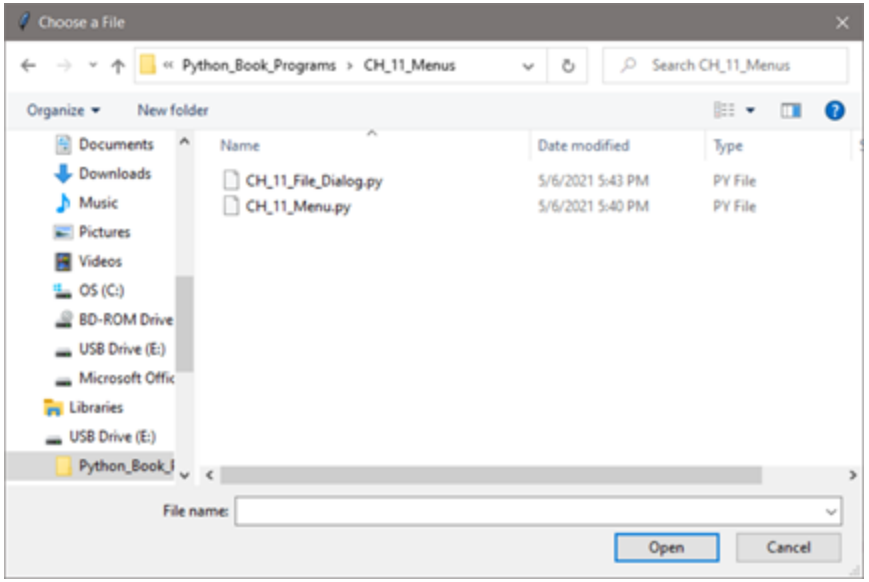
- File Dialogs – Open File
  - The tkinter module provides dialogs for handling files
    - Using them requires the specific import statement shown below

```
import tkinter
from tkinter import filedialog
```

```
filename = filedialog.askopenfilename(title='Choose a file.')
infile = open(filename,'r')
```

Chapter 7 File Operations

- File Dialogs – Open File Dialog
  - When the dialog appears, the default directory is the directory where the program is running



Chapter 7 File Operations

- File Dialogs – Open File
  - When a file is selected, the dialog returns a string containing the full path to the file including the name of the file
    - The string is used to open the file

```
filename = filedialog.askopenfilename(title='Choose a file.')
infile = open(filename,'r')
```

Chapter 7 File Operations

- File Dialogs
  - Tkinter provides Save and Save As dialogs among others

```
import tkinter.filedialog

tkinter.filedialog.asksaveasfilename()
tkinter.filedialog.asksaveasfile()
tkinter.filedialog.askopenfilename()
tkinter.filedialog.askopenfile()
tkinter.filedialog.askdirectory()
tkinter.filedialog.askopenfilenames()
tkinter.filedialog.askopenfiles()
```

Chapter 7 File Operations