Introduction to Programming and Problem Solving

Week 14: File Handling

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

codingissweet@gmail.com

Content

- Introduction file handling
- Need of File Handling
- Text Input and Output,
- The seek() Function
- Binary Files
- Accessing and Manipulating Files and Directories on a Disk
- Etc

Introduction

- A file is an object on a computer that stores data, information, settings, or commands used with a computer program. A computer has three types of files: application files, data files, and system files. [2]
- A file can also be thought of as a collection of records. A record is a group of related data items. These data items may contain information related to students, employees, customers, etc. In other words, a file is a collection of numbers, symbols and text and can be considered a stream of characters. [3]

Characteristics of a File:

- **Name:** Each file is identified by a unique name, which allows the operating system to distinguish one file from another.
- **Data:** A file contains information in the form of bytes. This data can be text, images, videos, executable programs, or any other kind of digital content.

Introduction+

- **Location:** Files are stored in directories or folders within a file system. Each file has a specific location in the file structure.
- **Attributes:** Files possess various attributes such as permissions, creation date, size, type, and more, which provide information about the file and regulate its access.

File Types:

- **Text Files:** Contain human-readable text, often encoded in formats like ASCII or UTF-8.
- **Binary Files:** Contain data that isn't in human-readable form and might include images, executables, databases, etc.
- **Program Files:** Contain instructions that a computer can execute, including software applications or scripts.

Introduction++

File Operations:

- **Creation:** Files can be created by users, applications, or the operating system.
- **Reading:** Data can be retrieved from files by reading their content using appropriate software.
- **Writing:** New information can be added or existing information modified within a file.
- **Deletion:** Files can be removed or deleted from the file system.

File handling in python

- File handling in programming involves managing files, performing operations like reading, writing, creating, and manipulating files on a computer system.
- In Python, file handling is a crucial aspect, allowing you to work with files for data storage, data retrieval, and other operations

Advantages of File Handling[1]

- **Versatility**: File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.

Advantages of File Handling+

- **Flexibility**: File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files, etc.), and to perform different operations on files (e.g. read, write, append, etc.).
- **User–friendly**: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.
- **Cross-platform**: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling[1]

- **Error-prone:** File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks:** File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity:** File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.
- **Performance:** File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations

File Modes

- 'r': Read (default mode). Opens a file for reading.
- 'w': Write. Opens a file for writing, truncating the file first.
- 'a': Append. Opens a file for writing, appending to the end of the file.
- 'b': Binary mode. (e.g., 'rb' or 'wb') for working with binary files.
- '+': Reading and writing. (e.g., 'r+' or 'w+')

Reading from and writing to a File

- You can read the contents of a file using various methods:
- read(): Reads the entire file.
- readline(): Reads a single line from the file.
- readlines(): Reads all lines in a file and returns them as a list.

Writing to a File

- To write content to a file, use the write() method.

```
1.# Open a file in write mode
2.file = open('filename.txt', 'w')
3.file.write("Hello, this is written to the file!")
4.file.close()
```

- Always close the file after performing operations to free up system resources.
- file.close()

Writing Text to a File

- Writing text to a file in Python involves opening the file in write mode and using the write() method to add content to the file.
- Steps to Write Text to a File:
- Open a File in Write Mode:

Writing Text to a File+

- Use the open() function with the file path and the mode 'w' to open the file for writing. If the file doesn't exist, it will be created. If it does exist, it will be truncated, meaning the existing content will be erased.
- with open('filename.txt', 'w') as file:
- # Perform writing operations
- Write Text to the File:
- Use the write() method to add content to the file.

```
1. with open('filename.txt', 'w') as file:
2.     file.write("This text will be written to the file.")
```

Writing Text to a File++

- Example with User Input:
- You can take user input and write it to a file.
- user_text = input("Enter text to write to the file: ")
- with open('output.txt', 'w') as file:
- file.write(user_text)

Appending to a File:

Writing Text to a File+++

- If you want to add content without erasing the existing content, open the file in append mode ('a').

```
1. with open('filename.txt', 'a') as file:
2.     file.write("This text will be appended to the file.")
```

- **Closing a File**

- Closing a file after you've finished performing operations on it is crucial in Python. It ensures that system resources are freed up and the file is properly saved and closed.

Writing Text to a File+++

- While Python automatically closes files at the end of the program, explicitly closing them is good practice.
- Closing a File Using the close() Method:
- After performing the necessary operations on a file, you can close it using the close() method.

```
file = open('filename.txt', 'w')
# Perform operations
file.close()
```

Using 'with' Statement for Automatic Closure

➢The with statement is recommended for handling files. It automatically closes the file after the suite inside the with block finishes executing, even in the case of exceptions or errors.

```
with open('filename.txt', 'w') as file:
    # Perform operations
# File is automatically closed outside the 'with' block
```

➢Closing the file is important because it releases the resources associated with the file. Not closing files can lead to issues like resource leaks, especially when dealing with a large number of files or working on systems with limited resources.

Writing Numbers to a File

- Writing numbers to a file in Python involves converting numerical data to a string format and then writing that string to the file.

Converting Numbers to Strings:

- Before writing numbers to a file, you need to convert them to strings using functions like str().

```
1.number = 42
2.number_as_string = str(number)
```

- Once the number is converted to a string, you can write it to a file using the write() method.

```
1.  with open('numbers.txt', 'w') as file:
2.      number = 42
3.      file.write(str(number))
```

Writing multiple numbers to a file

- If you have multiple numbers or data that you want to write to a file, you can concatenate them as strings before writing or write them individually with appropriate formatting.
- Example Writing Multiple Numbers to a File:
- numbers = [1, 2, 3, 4, 5]

```
1. with open('numbers.txt', 'w') as file:
2.     for number in numbers:
       # Writing each number on a new line
3.         file.write(str(number) + '\n')
```

Reading Text from a File

- Reading text from a file in Python involves opening the file in read mode and using methods to access and process the file's content.
- Steps to Read Text from a File:
- Open a File in Read Mode:
- Use the open() function with the file path and the mode 'r' to open the file for reading.

```
1. with open('filename.txt', 'r') as file:
2.     # Perform reading operations
```

Reading the File Content:

- read() Method: Reads the entire file content.

```
1. with open('filename.txt', 'r') as file:
2.     content = file.read()
3.     print(content)
```

- readline() Method: Reads a single line from the file.

```
1. with open('filename.txt', 'r') as file:
2.     line = file.readline()
3.     print(line)
```

readlines() Method

➢readlines() Method: Reads all lines in a file and returns them as a list.

```
1. with open('filename.txt', 'r') as file:
2.     lines = file.readlines()
3.     for line in lines:
4.         print(line)
```

➢Example to Read a File and Display its Content:

```
1. with open('example.txt', 'r') as file:
2.     content = file.read()
3.     print(content)
```

Reading Numbers from a File

- Reading numbers from a file in Python involves reading the text data from the file and converting it back into numerical format if needed.

Steps to Read Numbers from a File:

- Open a File in Read Mode:
- Use the open() function with the file path and the mode 'r' to open the file for reading.
- with open('numbers.txt', 'r') as file:
- # Perform reading operations

Reading the File Content and Converting to Numbers

- Using read() and split(): Read the content of the file and split it into individual elements, assuming the numbers are separated by whitespace or specific delimiters.

```
with open('numbers.txt', 'r') as file:
    content = file.read()
    numbers = content.split()  # Splits content into a list of strings
    numbers = [int(num) for num in numbers]  # Convert strings to integers
    print(numbers)
```

- Looping Through Lines: If numbers are stored line by line, read each line and convert it to numbers.

```
with open('numbers.txt', 'r') as file:
    numbers = [int(line) for line in file.readlines()]
    print(numbers)
```

Handling Decimal Numbers

- Handling Decimal Numbers: If you're dealing with decimal numbers, use float() instead of int() to convert to floating-point numbers.
- Example Reading Numbers from a File:
- Assuming a file 'numbers.txt' contains a series of integers separated by spaces or on individual lines:
- 5 10 15 20 25

```
1. with open('numbers.txt', 'r') as file:
2.     content = file.read()
3.     numbers = [int(num) for num in content.split()]
4.     print(numbers)
```

Reading Multiple Items on one Line

- If your file contains multiple items on a single line, such as numbers or any other data, you can read and parse them individually.
- Reading Multiple Items from a Single Line:
- Assuming your file has a structure like this:
- 5 10 15 20 25
- You can read this line and parse the numbers individually:

```
1. with open('numbers.txt', 'r') as file:
2.     line = file.readline()  # Read the line
3.     numbers = line.split()  # Split the line into individual items
4.     numbers = [int(num) for num in numbers]  # Convert strings to integers
5.     print(numbers)
```

Appending Data

- Appending data to a file in Python involves opening the file in append mode and adding new content to the existing file without overwriting its current contents.
- Appending Data to a File:
- Open a File in Append Mode:
- Use the open() function with the file path and the mode 'a' to open the file for appending.

```
1. with open('filename.txt', 'a') as file:
2.     # Perform appending operations
```

Appending Data+

- Appending Data to the File:\

```
1. with open('filename.txt', 'a') as file:
2.     file.write("New content to be appended\n")
```

- Example Appending Lines to a File:

- Assuming 'filename.txt' contains some existing content:

```
1.  with open('filename.txt', 'a') as file:
2.      file.write("This text will be appended to the file.\n")
3.      file.write("Another line appended to the file.\n")
```

- This will add the new content to the end of the existing file without erasing the content that's already present.

Exception handling in file

- Exception handling in file operations is crucial as it helps manage errors that can occur while reading, writing, or manipulating files. Python provides a mechanism to catch and handle these errors using try-except blocks.

Common File Handling Exceptions:

- **FileNotFoundError:** Raised when a file does not exist.
- **PermissionError:** Occurs when the program doesn't have the required permissions to access or modify the file.
- **IOError:** General I/O error that may arise during file operations.

Using Try-Except Blocks:

```
try:
    with open('filename.txt', 'r') as file:
        content = file.read()
        # Perform operations with the file content
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied to access the file.")
except IOError as e:
    print("An I/O error occurred:", e)
```

Handling Exceptions Generically:

```
try:
    # File handling operations
except Exception as e:
    print("An error occurred:", e)
```

TEXT INPUT AND OUTPUT

- Text input and output in Python are fundamental aspects of file handling and working with textual data.
- Input can come from various sources:
- Keyboard Input
- The input() function allows users to input text directly into a program. For instance:

```
1. user_input = input("Enter something: ")
2. print("You entered:", user_input)
```

Reading from Files

- Reading text from files involves opening a file in read mode and then using methods like read(), readline(), or readlines() to access the file's content.

```
1. with open('filename.txt', 'r') as file:
2.     content = file.read()
3.     print(content)
```

Need for file handling

- File handling is an essential aspect of programming, enabling you to store and retrieve data from files. It plays a crucial role in various programming tasks, including:
- **Data Storage and Persistence:** File handling allows you to save data generated by your programs, ensuring its persistence even after the program terminates. This enables you to access and reuse data across different program executions.
- **Input and Output Operations**: Programs often interact with the user by reading input from files and displaying output to files. File handling facilitates this interaction, allowing you to process large amounts of data efficiently.

Need for file handling+

- **Data Exchange and Archiving**: File handling enables you to exchange data between different programs or applications, regardless of their programming language or platform. It also facilitates data archiving for long-term storage and retrieval.
- **Error Handling and Logging**: File handling is crucial for logging program errors and events, providing valuable insights into program behavior and facilitating debugging.
- **Web Development**: File handling is essential for web development, enabling you to manage HTML, CSS, JavaScript, and other web page components. It also allows you to handle file uploads and downloads.

Need for file handling++

- **Database Interactions**: Many programming environments interact with databases using file-based interfaces. File handling facilitates communication with these databases, enabling data manipulation and retrieval.
- **Data Analysis and Visualization**: File handling is widely used in data analysis and visualization tools, allowing you to import, process, and analyze data from various sources, such as CSV files, Excel spreadsheets, and text files.
- **Machine Learning and Artificial Intelligence**: File handling plays a critical role in machine learning and AI applications, enabling you to load training and testing data, save trained models, and export predictions.

Need for file handling++

- **Configuration Management**: Many applications store configuration settings in files, enabling runtime configuration changes without modifying the program's code. File handling allows you to read and update these configuration files.
- **Software Deployment and Packaging**: File handling is essential for packaging software applications, allowing you to bundle program files, configuration files, and documentation into a distributable package.
- **Backup and Recovery** Files are vital for creating backups of critical data. They facilitate the recovery of information in case of system failures or data loss

The seek() function

- The seek() function is a built-in function in Python that is used to change the position of the file pointer within a file. The file pointer is a marker that indicates the current position in the file, and it is used to read or write data from that point.
- The seek() function takes two arguments:
- offset: A number representing the new position of the file pointer.
- whence: An optional argument that specifies the reference point for the offset. The possible values for whence are:
- 0: The beginning of the file
- 1: The current position of the file pointer
- 2: The end of the file

The seek() function+

For example, the following code seeks to the 10th byte of the file myfile.txt:

```
f = open("myfile.txt", "r")
f.seek(10)
```

This code will then read data from the file starting at the 10th byte.

The seek() function can also be used to move the file pointer backward. For example, the following code seeks to the beginning of the file myfile.txt:

```
f = open("myfile.txt", "r")
f.seek(0, 0)
```

This code will then read data from the file starting at the beginning.

The seek() function++

The seek() function is a powerful tool for working with files in Python. It can be used to read or write data from any position in a file, and it is essential for tasks such as reading large files or modifying specific portions of a file.

Here is an example of how to use the seek() function to read the contents of a file in reverse:

```
1. def read_file_in_reverse(filename):
2.     with open(filename, "r") as f:
3.         f.seek(0, 2)  # Seek to the end of the file
4.         line = f.readline()  # Read the last line of the file
5.         while line:
6.             print(line)
7.             f.seek(-len(line), 1)  # Seek back the length of the line
8.             line = f.readline()
read_file_in_reverse("myfile.txt")
```

Code Explanation

- Defining the Function: The function read_file_in_reverse takes a filename as an argument.
- Opening the File: It opens the file in read mode ("r") using a with statement, ensuring proper file handling.
- Moving to the End of the File: f.seek(0, 2) seeks the end of the file by using the seek() method.
- The second argument 2 indicates moving to the end of the file.
- Reading the Last Line: It reads the last line of the file using f.readline() and stores it in the variable line.
- Printing in Reverse Order: The code enters a while loop that continues as long as line is not an empty string (which would indicate the beginning of the file).

Code Explanation+

- Inside the loop, it prints the current line using print(line).
- Moving to the Previous Line: f.seek(-len(line), 1) seeks backward by the length of the current line. This step is important to move the file pointer back to the beginning of the previous line. It uses the second argument 1 to indicate the relative position for seeking backward.
- Reading the Previous Line: It reads the line before the current one with line = f.readline().
- Invoking the Function: The code calls the function read_file_in_reverse("myfile.txt"), passing the filename "myfile.txt" to read its content in reverse order.
- This code essentially reads a file from the end to the beginning, printing its content line by line in reverse order.

Binary Files

- Binary files are computer files that store data in a binary format, which means that the data is represented using sequences of 0s and 1s. This is in contrast to text files, which store data using characters that can be read and understood by humans.
- Binary files are used to store a wide variety of data, including:
- Executable programs: These are files that contain the machine code that can be executed by a computer's processor.
- Images: These are files that store digital images, such as photographs and graphics.
- Audio: These are files that store digital audio, such as music and voice recordings.

Binary Files+

- Video: These are files that store digital video, such as movies and TV shows.
- Data: These are files that store other types of data, such as scientific data, engineering data, and financial data.
- Binary files are typically not human-readable, which means that they cannot be opened and read directly in a text editor. Instead, they must be opened and read using a program that understands the binary format of the file.
- For example, a JPEG image file can be opened and read using a photo editing program, such as Adobe Photoshop or GIMP. An MP3 music file can be opened and read using a music player, such as iTunes or VLC Media Player.

Advantages of using binary files:

- **Smaller file size**: Binary files are typically smaller than text files, because they do not store the formatting information that is necessary for human-readable text. This can be important for applications that need to store large amounts of data, such as video files.
- **Faster processing**: Binary files can be processed more quickly than text files, because the computer does not need to convert the data from a human-readable format to a machine-readable format. This can be important for applications that need to process large amounts of data in real time, such as audio and video applications.
- **More accurate representation of data**: Binary files can store data more accurately than text files, because they do not need to use approximations to represent characters. This can be important for applications that need to store precise data, such as scientific data and engineering data.

Disadvantages of using binary files:

- Not human-readable: Binary files are not human-readable, which means that they cannot be opened and read directly in a text editor. This can make it difficult to debug problems with binary files.
- Requires special software: Binary files require special software to open and read them. This can be a problem if you do not have the right software available.
- More susceptible to corruption: Binary files are more susceptible to corruption than text files, because a single bit error can corrupt the entire file.

Handling binary files

- Handling binary files involves reading, writing, and manipulating data stored in a binary format. Binary files are typically used to store data that cannot be represented in a human-readable format, such as images, audio, and executable programs.

Reading Binary Files

- To read a binary file, you will need to open the file in binary mode and then use a function like read() or readline() to read the data. The data will be returned as a sequence of bytes, which you can then convert to the desired data type.
- For example, to read an image file, you could use the following code:
- with open('image.jpg', 'rb') as f: data = f.read()

Handling binary files+

- This code will read the entire contents of the image file into a byte array. You can then use a library like Pillow to convert the byte array into an image object.

Writing Binary Files

- To write a binary file, you will need to open the file in binary mode and then use a function like write() to write the data. The data must be passed as a sequence of bytes.
- For example, to write an image file, you could use the following code:
- `with open('image.jpg', 'wb') as f: f.write(data)`
- This code will write the byte array data to the image file.

Manipulating Binary Files

- You can also manipulate binary files by reading and writing specific bytes. For example, you could change the color of a pixel in an image file by changing the corresponding bytes in the file.

```
1. with open('image.jpg', 'rb') as f:
2.     data = f.read()
   # Change the color of the pixel at (100, 100) to red
3. data[100 * 3 + 100 * 3] = 255
   with open('image.jpg', 'wb') as f:
4.     f.write(data)
```

Tips for handling binary files

- Always open binary files in binary mode. This will prevent the file from being corrupted.
- Use the appropriate functions for reading and writing binary data. Do not try to read binary data as text, or vice versa.
- Be careful when manipulating binary data. A single bit error can corrupt the entire file.
- Use a library like Pillow or PyAudio if you are working with images or audio files. These libraries provide a higher-level interface for working with these types of files.

Accessing and Manipulating Files and Directories on a Disk

➢ Accessing and manipulating files and directories on a disk is a fundamental task in computing. It involves creating, reading, writing, deleting, and moving files and directories. These operations are essential for managing data, organizing files, and interacting with the underlying storage system.

Creating Files and Directories

➢ To create a new file, you can use the open() function with the w or a mode, depending on whether you want to write a new file or append to an existing one. For directories, you can use the mkdir() function

```
1. with open('myfile.txt', 'w') as f:
2.     f.write('This is a new file.')
   # Create a new directory named "mydir"
3. import os
4. os.mkdir('mydir')
```

Accessing and Manipulating Files and Directories on a Disk+

Reading Files

➢ To read the contents of a file, you can use the read() function. The read() function can read the entire file or a specified number of bytes.

```
1. with open('myfile.txt', 'r') as f:
2.     contents = f.read()
3.     print(contents)
   # Read the first 10 bytes of "myfile.txt"
4. with open('myfile.txt', 'r') as f:
5.     data = f.read(10)
6.     print(data)
```

Accessing and Manipulating Files and Directories on a Disk++

Writing to Files

- To write data to a file, you can use the write() function.
- The write() function takes a string as an argument and writes it to the file.

```
1. with open('myfile.txt', 'a') as f:
2.     f.write('This is a new line.\n')
```

Accessing and Manipulating Files and Directories on a Disk++

➢Deleting Files and Directories

➢To delete a file, you can use the remove() function. To delete a directory, you can use the rmdir() function.

```
1. import os
2. os.remove('myfile.txt')
3.
   # Delete the directory "mydir"
4. os.rmdir('mydir')
```

Accessing and Manipulating Files and Directories on a Disk+++

Moving and Renaming Files and Directories

- To move a file or directory, you can use the rename() or move() function.
- The rename() function changes the name of a file or directory, while the move() function changes both the name and location of a file or directory.

```
1. import os
2. os.rename('myfile.txt', 'mynewfile.txt')
   # Move the directory "mydir" to the parent directory
3. os.move('mydir', '..')
```

Accessing and Manipulating Files and Directories on a Disk++++

Navigating the File System

➢To navigate the file system, you can use the os.chdir() function to change the current working directory. You can also use the os.listdir() function to list the contents of a directory.

```
1. import os
2. # Change the current working directory to the "mydir" directory
3. os.chdir('mydir')
4. # List the contents of the current directory
5. contents = os.listdir('.')
6. print(contents)
```

Files and Dictionary

Let us look at a simple project that demonstrate the practical use of files with dictionaries

Count words in the file

```
1.  fn = input('Enter the file name: ')
2.  try:
3.      fh = open(fn)
4.  except:
5.      print('File cannot be opened:', fn)
6.      exit()
7.
    counts = {}
8.  for ln in fh:
9.      words = ln.split()
10.     for w in words:
11.         if w not in counts:
12.             counts[w] = 1
13.         else:
14.             counts[w] += 1
15. print(counts)
```

```
16. for key, value in counts.items():
17.     if value>10:
18.         print(f'{key} : {value}')
```

Explanation

- **User Input:** The code starts by asking the user to input a file name. It uses the input() function to prompt the user for the file name. The entered file name is stored in the variable fn.
- **File Opening:** It tries to open the file specified by the user input using the open() function within a try-except block. If the file is not found or cannot be opened for any reason, it will print an error message stating that the file cannot be opened and then exits the program using exit().
- **Word Counting:** Assuming the file is successfully opened, the code initializes an empty dictionary called counts to store word counts.

Explanation+

- **Iteration through File Content:** It then proceeds to read the file line by line using a for loop: for ln in fh:. It splits each line into words using the split() method and stores them in the words variable.
- **Counting Words**: It iterates through each word in the words list. For each word, it checks if the word is already a key in the counts dictionary. If it's not, it adds the word as a key to the dictionary with a count of 1. If the word is already a key in the dictionary, it increments the count by 1.
- **Printing the Word Counts:** Finally, after iterating through the entire file, it prints the counts dictionary, which now contains each word in the file as a key and the number of times it appeared as the corresponding value.

File Handling best Practices

- **File Closure:** Always close the file after performing operations to avoid resource leaks.
- **Error Handling**: Implement error handling to manage exceptions during file operations.
- **File Paths**: Use appropriate file paths and handle file path issues (like using absolute or relative paths).
- **Context Managers (with statement):** Utilize the with statement to ensure that files are automatically closed after usage.

Summary

- As Week 14 concludes, it's time for a swift recap. This week has been dedicated to delving into file handling in Python. We've explored a spectrum of file operations, delved into the essentiality of file handling, and even touched upon its drawbacks.
- Throughout our sessions, we've generously shared a multitude of code examples, aiming to ensure a comprehensive understanding of these pivotal concepts.
- With the culmination of our lectures on programming introduction and problem-solving, I extend my heartfelt gratitude for your active participation. I sincerely hope you've attained a profound knowledge of programming and problem-solving. If not, I wish you the very best on your continued journey in problem-solving.

Reference

[1] (2023, June 27). *File Handling in Python*. GeeksforGeeks. Retrieved November 12, 2023, from https://www.geeksforgeeks.org/file-handling-python/

[2] (2023, October 1). *What is a File?* Computer Hope. Retrieved November 12, 2023, from https://www.computerhope.com/jargon/f/file.htm

[3] Kamthane, A. N., & Kamthane , A. A. (2018). *PROGRAMMING AND PROBLEM SOLVING WITH PYTHON* (p. 356). McGraw Hill Education (India) Private Limited.