**Module 9. File and Exception Handling**

**Learning Objectives**

* Understand and utilize various file modes in Python for reading, writing, and creating files.
* Implement best practices for file handling using the with statement and exception handling.
* Read and process files line by line or all at once using Python methods.
* Write to files, either appending or overwriting content, while ensuring proper resource management.
* Handle file-related errors gracefully using try-except blocks and custom exceptions.

1. **Understanding Files in Programming**

To a computer user, a file is a named set of information that is useful for a particular purpose. This could be a document, a picture, a song, or any other type of data. The filename itself may not be critically important, as long as the file is accessible and can retain the data over time, typically stored on long-term storage devices like hard disk drives.

For your programs, files are just as useful but require a bit more management to ensure that they work correctly.

File handling is an essential part of programming, enabling the reading, writing, and manipulation of files. In Python, the primary function used for file handling is ***open()***.

* 1. **The open() Function**

The ***open()*** function takes two parameters:

* **Filename**: The name of the file you want to open.
* **Mode**: Specifies the purpose of opening the file.

**File Modes**

Here are the different modes for opening a file:

* **Read Mode ("r")**:
  + Opens a file for reading.
  + Returns an error if the file does not exist.
  + Default mode.

|  |
| --- |
| f = open("demofile.txt", "r") |

* **Append Mode ("a")**:
  + Opens a file for appending (adding content to the end).
  + Returns an error if the file does not exist.

|  |
| --- |
| f = open("demofile.txt", "a") |

* **Write Mode ("w")**:
  + Opens a file for writing.
  + Creates the file if it does not exist.
  + Overwrites the file if it exists.

|  |
| --- |
| f = open("demofile.txt", "w") |

* **Create Mode ("x")**:
  + Creates the specified file.
  + Returns an error if the file exists.

|  |
| --- |
| f = open("demofile.txt", "x") |

**Text and Binary Modes**

In addition to the basic modes, you can specify how the file should be handled:

* **Text Mode ("t")**:
  + Default mode.
  + Opens the file in text mode.
* **Binary Mode ("b")**:
  + Opens the file in binary mode.
  + Useful for non-text files (e.g., images, executable files).
  1. **Reading a File in Python**

Reading files is a common operation in many programming tasks. Below is the basic way to read the contents of a file in Python, with some additional best practices.

**Basic File Reading**

Here's a simple example of how to open a file for reading and print its contents:

|  |
| --- |
| # Open a file for reading  f = open("demofile.txt", "r")  print(f.read())  f.close() |

(Explanation)

* **Open the File**: The ***open*** function is used to open the file "demofile.txt" in read mode ("r"). You need to ensure that the file "demofile.txt" must be located in the same folder where this script is run.
* **Read the Contents**: The ***read*** method reads the entire content of the file.
* **Print the Contents**: The ***print*** function prints the content to the console.
* **Close the File**: The ***close*** method closes the file, which is important to free up system resources.

**Best Practices**

Using the ***with*** statement is recommended for handling files because it ensures that the file is properly closed even if an error occurs during the file operation. Here's how you can do it:

|  |
| --- |
| # Open a file for reading using the 'with' statement  with open("demofile.txt", "r") as f:  content = f.read()  print(content)  # No need to explicitly close the file |

**Reading a File Line by Line**

If you want to read multiple lines, you can call *readline()* multiple times or use a loop. Here’s an example using a loop:

|  |
| --- |
| # Open a file for reading  with open("demofile.txt", "r") as f:  # Read and print lines one by one  while True:  line = f.readline()  if not line: # End of file  break  print(line.strip()) # Strip newline characters |

(Explanation)

* **Using with Statement**: Ensures the file is properly closed after its suite finishes.
* **Loop through Lines**: Use a ***while*** loop to read each line until the end of the file.
* **Check for End of File**: ***readline()*** returns an empty string when the end of the file is reached.
* **Print Each Line**: ***print(line.strip())*** prints the line, removing any leading or trailing whitespace including newline characters.

Or you can read the file line by line using the file object directly. Here’s an example:

|  |
| --- |
| # Open a file for reading line by line  with open("demofile.txt", "r") as f:  for line in f:  print(line.strip()) |

(Explanation)

* **Using with Statement**: Ensures the file is properly closed after its suite finishes, even if an exception is raised.
* **Reading Line by Line**: The ***for*** loop reads the file line by line.
* **Strip Whitespace**: ***strip()*** removes any leading and trailing whitespace, including the newline character at the end of each line.

**Reading All Lines into a List**

If you need to read all lines at once and store them in a list, you can use the ***readlines()*** method:

|  |
| --- |
| # Open a file for reading  with open("demofile.txt", "r") as f:  lines = f.readlines()  # Print each line from the list  for line in lines:  print(line.strip()) |

(Explanation)

* **Read All Lines**: ***readlines()*** reads all the lines in the file and returns them as a list.
* **Iterate Through List**: Use a for ***loop*** to iterate through the list and process each line.
  1. **Writing to an Existing File in Python**

To write to an existing file in Python, you can use the open() function with specific modes. The two most common modes for writing are:

* **Append Mode ("a")**: Appends content to the end of the file without removing the existing content.
* **Write Mode ("w")**: Overwrites any existing content in the file.

**Example: Appending to a File**

Here's an example of how to append content to an existing file:

|  |
| --- |
| def main():  # Open the file "demofile2.txt" in append mode  f = open("demofile2.txt", "a")    # Write content to the file  f.write("Now the file has more content!")    # Close the file  f.close()  # Open and read the file after appending  f = open("demofile2.txt", "r")    # Print the content of the file  print(f.read())    # Close the file  f.close()  if \_\_name\_\_=="\_\_main\_\_":  main() |

(Explanation)

* **Open the File in Append Mode**: Use ***open("demofile2.txt", "a")*** to open the file in append mode.
* **Write to the File**: The ***write()*** method appends the specified string to the end of the file.
* **Close the File**: Always close the file after writing to ensure all changes are saved and resources are freed.
* **Read and Print the File**: Reopen the file in read mode and print its content to verify that the new content has been appended.

**Example: Overwriting a File**

If you want to overwrite the existing content, you can use write mode ("w") instead:

|  |
| --- |
| def main():  # Open the file "demofile2.txt" in write mode  f = open("demofile2.txt", "w")  # Write content to the file, overwriting any existing content  f.write("This content will overwrite the existing content!")  # Close the file  f.close()  # Open and read the file after overwriting  f = open("demofile2.txt", "r")  # Print the content of the file  print(f.read())  # Close the file  f.close()  if \_\_name\_\_=="\_\_main\_\_":  main() |

(Explanation)

* **Open the File in Write Mode**: Use ***open("demofile2.txt", "w")*** to open the file in write mode.
* **Write to the File**: The ***write()*** method overwrites any existing content with the specified string.
* **Close the File**: Always close the file after writing to ensure all changes are saved and resources are freed.
* **Read and Print the File**: Reopen the file in read mode and print its content to verify that the old content has been overwritten.

**Best Practices**

* **Use with Statement**: Using the with statement ensures that files are properly closed after their suite finishes, even if an exception is raised.

|  |
| --- |
| with open("demofile2.txt", "a") as f:  f.write("Now the file has more content!")  with open("demofile2.txt", "r") as f:  print(f.read()) |

* **Error Handling**: Handle potential errors such as file not found or permission issues using try-except blocks.

|  |
| --- |
| try:  with open("demofile2.txt", "a") as f:  f.write("Now the file has more content!")  except IOError as e:  print(f"An error occurred: {e}") |

* 1. **Creating a New File in Python**

In Python, you can create a new file using the open() function with specific modes. Here’s how you can do it with different modes:

* **Create Mode ("x")**: Creates a new file and returns an error if the file already exists.
* **Append Mode ("a")**: Opens a file for appending and creates the file if it does not exist.
* **Write Mode ("w")**: Opens a file for writing and creates the file if it does not exist.

**Example: Creating a File Using Write Mode**

To create a file using the write mode, which will create the file if it does not exist:

|  |
| --- |
| def main():  # Open a file for writing (creates if it does not exist)  with open("newfile\_write.txt", "w") as f:  f.write("This file is created and written to in write mode.\n")  # Verify the file content  with open("newfile\_write.txt", "r") as f:  print(f.read())  if \_\_name\_\_=="\_\_main\_\_":  main() |

(Explanation)

* **Open the File in Write Mode**: Use ***open("newfile\_write.txt", "w")*** to open the file for writing. This will create the file if it does not exist.
* **Write to the File**: Use ***write()*** to add content to the file.
* **Close the File**: Always close the file after performing operations on it.
* **Read the File**: Reopen the file in read mode to verify its content.
  1. **Code Examples**

This code example demonstrates creating a file of usernames from a file that contains names.

|  |
| --- |
| def main():      try:          print("Creating a file of usernames from a file of names.")            # Get the file names          infileName = input("What file are the names in? ")          outfileName = input("What file should the usernames go in? ")            # Open the files          infile = open(infileName, 'r')          outfile = open(outfileName, 'w')            # Process each line of the input file          for line in infile:              # Get the first and last names from line              first, last = line.split()              # Create a username using slicing              uname = (first[0] + last[:7]).lower()                # Write it to the output file              print(uname, file=outfile)            # Close both files          infile.close()          outfile.close()            print("Usernames have been written to", outfileName)      except IOError as e:          print(f"An error occurred: {e}")  if \_\_name\_\_=="\_\_main\_\_":      main() |

1. **Exception Handling in Python**

Exception handling in Python allows you to manage errors and exceptional events gracefully. By using try-except blocks, you can ensure your program doesn't crash unexpectedly and can handle errors appropriately.

**Why Exception Handling?**

* **Graceful Degradation**: Prevents the program from terminating unexpectedly.
* **Resource Management**: Ensures resources are properly released (e.g., closing files, database connections).
* **User Feedback**: Provides informative messages to users when something goes wrong.

**Basic Exception Handling**

Here's a basic example of handling a ***FileNotFoundError*** when trying to open a missing file:

|  |
| --- |
| try:  myFile = open("missing.txt")  # Read in the 1st line only  oneLine = myFile.readline()  print("You have one chance to impress: ", oneLine)  myFile.close()  except FileNotFoundError as fnfe:  print("The file you attempted to open isn't present, blockhead!")  print("By the way, the error information is: ", fnfe, "(blockhead..)") |

(Explanation)

* **Try Block**: Contains the code that might raise an exception.
* **Except Block**: Handles specific exceptions. In this case, ***FileNotFoundError***.
* **Error Object**: The ***as*** keyword assigns the exception object to a variable (***fnfe***) for further inspection.

**Finally Block**

The ***finally*** block allows you to execute code whether or not an exception occurred. This is useful for cleanup actions like closing files or releasing resources.

|  |
| --- |
| try:  myFile = open("missing.txt")  # Read in the 1st line only  oneLine = myFile.readline()  print("You have one chance to impress: ", oneLine)  myFile.close()  except FileNotFoundError as fnfe:  print("The file you attempted to open isn't present, blockhead!")  print("By the way, the error information is: ", fnfe, "(blockhead..)")  finally:  print("You should perform any tasks that must be completed even if the try block abnormally ends here in the finally block...blockhead") |

**Using with Statement**

The with statement simplifies resource management by ensuring resources are automatically cleaned up. For file handling, it means you don't need to explicitly close the file.

|  |
| --- |
| with open("amateurlyrics.txt") as myFile:  # Read in the 1st line only  oneLine = myFile.readline()  print("You have one chance to impress: ", oneLine) |

**Custom Exceptions**

You can define your own exceptions by creating a new exception class. This is useful for handling specific errors in your application.

|  |
| --- |
| class MyException(Exception):  def \_\_init\_\_(self):  print("A new object of type MyException")  # Raising a custom exception  raise MyException |

(Explanation)

* **Custom Exception Class**: ***MyException*** extends the base ***Exception*** class.
* **Raising the Exception**: The ***raise*** keyword is used to trigger the exception.

**Example Scenario: File Handling with Proper Cleanup**

This example demonstrates how to handle file operations in Python with proper exception handling and cleanup. It uses the ***try-except-finally*** blocks along with the ***with*** statement for resource management.

|  |
| --- |
| try:  with open("example.txt", "r") as file:  content = file.read()  print(content)  except FileNotFoundError as e:  print("File not found:", e)  finally:  print("Finished attempting to read the file.") |

(Explanation)

* ***with* Statement**: Automatically handles closing the file, reducing the risk of resource leaks.
* **Specific Exception Handling**: The ***except*** ***FileNotFoundError*** block handles the specific case where the file is not found, providing a clear error message.
* **Cleanup with *finally***: Ensures that the cleanup code is executed regardless of the outcome of the ***try*** block.

**Example of Handling Multiple Specific Exceptions**

Here's how you can handle multiple specific exceptions within a single try block:

|  |
| --- |
| try:  with open("example.txt", "r") as file:  content = file.read()  print(content)  except FileNotFoundError as e:  print("File not found:", e)  except IOError as e:  print("IO error occurred:", e)  finally:  print("Finished attempting to read the file.") |

**Summary**

1. Files in programming are named sets of information stored on long-term storage devices like hard drives.
2. The ***open()*** function in Python is essential for file handling, requiring a filename and a mode to open the file.
3. In read mode ("***r***"), a file is opened for reading and an error is returned if the file does not exist.
4. Append mode ("***a***") opens a file for adding content to the end, returning an error if the file does not exist.
5. Write mode ("***w***") opens a file for writing, creating it if it does not exist and overwriting it if it does.
6. Create mode ("***x***") creates a new file and returns an error if the file already exists.
7. Text mode ("***t***") is the default mode, opening the file in text mode, while binary mode ("***b***") is used for non-text files.
8. The ***read()*** method reads the entire content of a file, which is then printed and the file is closed to free up resources.
9. Using the ***with*** statement for file handling ensures that the file is properly closed, even if an error occurs.
10. Reading a file line by line can be done using a loop with the ***readline()*** method or directly with a for loop.
11. The ***readlines()*** method reads all lines of a file at once and stores them in a list for further processing.
12. To write to an existing file, use append mode ("***a***") to add content without removing existing data, or write mode ("***w***") to overwrite the file.
13. Creating a new file can be done using write mode ("***w***"), append mode ("***a***"), or create mode ("***x***"), depending on the need.
14. Exception handling in Python, using ***try-except*** blocks, helps manage errors and prevent program crashes.
15. The ***finally*** block allows execution of code regardless of whether an exception occurred, ensuring proper cleanup actions.

**Programming Exercises**

1. **File Content Reverse Display**

Write a program that reads a file containing a series of strings and displays the content of the file in reverse order, starting from the last line to the first.

1. **File Tail Display**

Write a program that asks the user for the name of a file and displays the last five lines of the file’s contents. If the file contains less than five lines, it should display the file’s entire contents.

1. **Word Counter**

Write a program that asks the user for the name of a file. The program should read the file and display the total number of words in the file.

1. **Vowel Counter**

Assume a file containing a series of strings is named and exists on the computer’s disk. Write a program that counts the number of vowels (a, e, i, o, u) in the file and displays the count.

1. **Character Frequency**

Write a program that asks the user for the name of a file. The program should read the file and display the frequency of each character in the file.

1. **Word Frequency**

Write a program that asks the user for the name of a file. The program should read the file and display the frequency of each word in the file.

1. **Longest Line**

Write a program that asks the user for the name of a file. The program should read the file and display the longest line in the file.

1. **Unique Words**

Write a program that asks the user for the name of a file. The program should read the file and display all unique words in the file.

1. **Duplicate Line Removal**

Write a program that asks the user for the name of a file. The program should read the file and create a new file that contains the same content but with all duplicate lines removed.

1. **File Content Search**

Write a program that asks the user for the name of a file and a search term. The program should read the file and display all lines that contain the search term.