

File Handling

Unit-6

Contents

Introduction

File Opening Models

Reading and Writing Files

The os Module and Common Functions

The with Statement.

Introduction

File handling is a fundamental concept in programming that allows you to interact with file in your computer's storage.

Python provide a robust set of tools for working with files, making it easy to read , write and manipulate data.

A file is a named location used for storing data. For example, main.py is a file that is always used to store Python code.

Python provides various functions to perform different file operations, a process known as File Handling.

This process involves two main operations:

- **File I/O (Input/Output):**
 - Reading data from a file (Input)
 - Writing data to a file (Output)
- **File Objects:** Python represents files as objects when you open them, allowing you to interact with them using methods provided by the file object.

Why is File Handling Operation Used?

It offers data persistence by allowing you to store the data beyond the run time.

It allows easy input and output of data which makes so many operations possible.

The files can be organized as it also offers easy-going storage and retrieval of data.

It facilitates data processing, making analysis and cleaning of data hassle-free.

It also provides easy sharing and communication of data.

In Python, file handling works in three ways which are:

1. Opening the file
2. Working on a file, that is, reading or writing
3. Closing the file

Example

```
# Open the file named "file.txt" in reading mode
file = open("file.txt", "r")

# Read and print each line from the file
for line in file:
    print(line.strip()) # Using .strip() to remove leading/trailing whitespace

# Close the file
file.close()
```

Opening Files in Python

This is the first step that is mandatory in file handling. You can not proceed to perform other functions like reading or writing data when you have not opened the file. The in-built open() function is used to get access to this step.

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

File Opening Modes

Mode	Function
r	read-only mode
w	write only mode
a	append operation in an existing file
r+	read and write data mode in file
w+	write and read data mode in file
a+	append and read data in a file

We can also specify if the file should be handled as binary or text mode :

- 'b' - Binary mode: Opens the file in binary mode. Used with other modes like 'rb' (read binary) or 'wb' (write binary).
- 't' - Text mode (default mode): Opens the file in text mode. This is the default if no mode is specified.

Text File

Opening file and reading from file

To open a file we can use the built-in function named `open()` and it returns a file object.

- `file = open("abc.txt", "r")`
- `file = open("d:\\nccs\\python\\abc.txt", "r")`
- `file = open("abc.txt")`

Note: Since both “r” and “t” both are default values for opening modes so just mentioning the file’s name is sufficient to open for reading mode.

- `file = open("abc.txt")`

Reading file content

Reading the Entire File with `read()`: The `read()` method reads the entire content of the file as a single string.

Syntax:

- `file_object.read(size=-1)`

`size`: Optional. The number of bytes to read. Default is -1, which means the entire file.

Reading Line by Line with readline(): The readline() method reads one line from the file at a time.

Syntax

- **file_object.readline(size=-1)**

size: Optional. The number of bytes to read. Default is -1, which means the entire line.

Reading All Lines with `readlines()`: The `readlines()` method reads all the lines of a file and returns them as a list of strings.

Syntax:

- `file_object.readlines(hint=-1)`

`hint`: Optional. If specified, only reads `hint` number of lines.

`print(file.read)` -> read entire text by default. But we can tell how many characters to read

`print(file.read(20))` -> read 20 characters

`print(file.readline())` -> read a single line

`print(file.readline())`

`print(file.readline())` -> read two lines

```
file= open('example.txt', 'r')  
  
for line in file:  
  
    print(line, end="")
```

This iterates over each line of example.txt and prints it.

The with statement for Safe file handling

The with statement in Python is used for resource management and ensures that resources are properly cleaned up after use.

When dealing with file operations, the with statement ensures that a file is properly closed after its suite finishes, even if an exception is raised during the process.

Syntax:

```
with open(file, mode) as file_object:
```

```
    # Perform file operations
```

file: The name or path of the file to open.

mode: The mode in which to open the file (e.g., 'r' for reading, 'w' for writing).

file_object: The variable to which the file object is assigned.

Benefits

- Automatic Resource Management: Ensures that the file is closed automatically after the block of code is executed, which prevents resource leaks.
- Exception Handling: Properly handles exceptions and ensures that the file is closed even if an error occurs within the block.
- Concise and Readable Code: Simplifies the code by eliminating the need to explicitly close the file with `file.close()`.
- Safety: Reduces the risk of errors related to manual file management, such as forgetting to close the file.

Example: Reading a File

Without “with” Statement:

try:

```
file = open('example.txt', 'r')
```

```
content = file.read()
```

```
print(content)
```

finally:

```
file.close()
```

With “with” Statement:

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

Explanation:

- Without with Statement: We explicitly open the file using `open()`, read its content, and then close it using `file.close()` inside a finally block to ensure the file is closed even if an exception occurs.
- With with Statement: We open the file using `open()` within the `with` statement. The file is automatically closed after the block is executed, even if an exception occurs.

Write to a file

To write data in a file first we need to open a file in “a” or in “w” mode

Then, we can use

```
file.write(string) // to write string in a file
```

Using the write() method:

This method allows you to write a string to a file. Here's an example:

```
with open("myfile.txt", "w") as file:
```

```
    file.write("Hello, World!")
```

In this example:

`open("myfile.txt", "w")` opens the file "myfile.txt" in write mode ("w"). This mode will create a new file if it doesn't exist, and overwrite any existing content.

`with` statement ensures the file is automatically closed after the indented block, even if there are exceptions.

`file.write("Hello, World!")` writes the string "Hello, World!" to the file.

```
# Writing to a new file (w mode)
with open("newfile.txt", "w") as file:
    file.write("This is a new file.\n")
    file.write("Adding another line.")
```

```
# Appending to an existing file (a mode)
with open("data.txt", "a") as file:
    file.write("\nNew data appended.)
```

```
# Writing with newline character
content = "Line 1\nLine 2\n"
with open("myfile.txt", "w") as file:
    file.write(content)
```

Opening Modes:

"w" (write): This mode creates a new file if it doesn't exist, and overwrites any existing content. Be cautious as existing data will be lost.

"a" (append): This mode opens the file for appending content. If the file doesn't exist, it will be created. Any data written using write() will be added to the end of the file.

"x" (exclusive creation): This mode creates a new file and fails if the file already exists.

Creating a file

Using the open() function with specific modes:

The open() function is the workhorse for file handling in Python. By specifying a mode when opening the file, you can control how the file is created or accessed.

`open("myfile.txt", "w") // create a file if it does not exists`

`open("abc.txt", "x") // creates a file`

Important points:

Always be cautious with "w" mode, as it overwrites existing files.

Use "x" mode if you specifically need to create a new file and want to avoid overwriting existing data (but handle potential exceptions).

Consider using "a" mode for appending content to an existing file.

Remember to properly close the file using the with statement or close() method.

Binary Files

Binary files are a fundamental concept in Python for storing and working with data that isn't human-readable text.

They store information as a sequence of bytes (0s and 1s), which can represent various data types like images, audio, compressed data, or custom program data.

Opening a File in Binary Mode

The `open()` function is used to open files, but you need to specify binary mode for binary files.

`"rb"` (read binary): Opens a file for reading binary data.

`"wb"` (write binary): Opens a file for writing binary data (creates a new file if it doesn't exist).

`"ab"` (append binary): Opens a file for appending binary data (creates a new file if it doesn't exist).

Example: Opening a binary image file:

```
# Reading a binary file  
with open('example.bin', 'rb') as file:  
    binary_data = file.read()  
    print(binary_data)
```

Writing to Binary Files:

Use the `write()` method to write bytes to a binary file opened in write or append mode.

Data needs to be converted to bytes before writing. Here are common methods:

Integers: Convert integers to bytes using `int.to_bytes()`.

Floats: Convert floats to bytes using `struct.pack()` (more complex for different float sizes).

Strings: Encode strings to bytes using `bytes.fromhex()` for hexadecimal strings or `string.encode()` for specific encodings (e.g., UTF-8).

Lists/Arrays: Convert elements to bytes individually and write them sequentially.

```
# Writing to a binary file  
  
data_to_write = b'This is some binary data'  
  
with open('example.bin', 'wb') as file:  
  
    file.write(data_to_write)
```

In this example, `example.bin` is opened in write binary mode. The `file.write()` method writes the bytes data to the file.

Appending to Binary Files

To append to a binary file, you use the ab mode. The data appended to the file must be in the form of bytes.

```
# Appending to a binary file  
  
data_to_append = b'\nThis is additional binary data'  
  
with open('example.bin', 'ab') as file:  
  
    file.write(data_to_append)
```

Reading from Binary Files:

Use the `read()` method to read a specific number of bytes (as an integer) or read the entire file at once.

The returned data is a `bytes` object, which needs interpretation based on the data structure.

Similar to writing, you might need to convert `bytes` back to their original data types.

```
# Reading a binary file  
with open('example.bin', 'rb') as file:  
    binary_data = file.read()  
    print(binary_data)
```

In this example, `example.bin` is opened in read binary mode. The `file.read()` method reads the entire file and returns the data as a bytes object.

Seeking in Binary files

Seeking in binary files allows you to move the file pointer to different positions within the file for reading or writing data. This is particularly useful when dealing with large files where you need to access specific parts without reading the entire file into memory.

In Python, you use the `seek()` method to move the file pointer, and the `tell()` method to find out the current position of the file pointer.

The seek() Method

The seek() method is used to move the file pointer to a specific position in the file. The syntax is:

```
file.seek(offset, whence)
```

offset: The number of bytes to move the pointer.

whence: The reference position from where offset is added. It can be:

0: Beginning of the file (default)

1: Current file position

2: End of the file

The tell() Method

The tell() method returns the current position of the file pointer.

The OS Module and Common Functions

The `os` module in Python provides a way of using operating system-dependent functionality, such as reading or writing to the file system, interacting with the environment, and managing processes. It's a powerful module for handling various tasks related to the operating system.

Common Functions in the os Module

`os.path.exists(path)`: This function checks if a path exists as a file or directory.

`os.remove(path)`: This function deletes a file at the specified path.

`os.makedirs(path)`: This function creates a directory and any necessary subdirectories to complete the path.

`os.rename()`: This function renames a file or directory from original source to new destination. `os.rename(src, dest)`

`os.path.exists(path)`: This function checks if a path exists as a file or directory.

`os.path.isfile(path)`: This function checks if a path exists as a regular file.

`os.path.isdir(path)`: This function checks if a path exists as a directory.

`os.rmdir(path)`: This function removes an empty directory.