TEXT FILE HANDLING IN PYTHON

Text file handling in Python involves operations such as reading from, writing to, and manipulating text files using Python's built-in file handling mechanisms. Here's a detailed overview:

Opening a File:

You can open a file using the built-in `open()` function. It takes two arguments - the file path and the mode ('r' for reading, 'w' for writing, 'a' for appending, and 'r+' for both reading and writing). If the file does not exist, it will be created.

```
# Opening a file in read mode
file = open("example.txt", "r")
```

Reading from a File:

After opening a file in read mode, you can read its contents using various methods like `read()`, `readline()`, or `readlines()`.

```
# Reading the entire file
content = file.read()

# Reading one line at a time
line = file.readline()

# Reading all lines into a list
lines = file.readlines()
```

Difference between read(), readline() and readlines()

In Python, when working with text files, `read()`, `readline()`, and `readlines()` are methods used to read data from a file, but they differ in their behavior and what they return:

1. `read()`:

- Reads the entire contents of the file as a single string.

- If you don't specify the number of bytes to read, it will read the entire file.

- Example:

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

2. `readline()`:

- Reads a single line from the file.

- Each time you call `readline()`, it reads the next line from the file.

- Example:

```
with open('file.txt', 'r') as file:
    line1 = file.readline()  # Reads the first line
    line2 = file.readline()  # Reads the second line
```

3. `readlines()`:

- Reads all the lines of a file and returns them as a list of strings.

- Each string in the list represents a single line from the file.

- Example:

```
with open('file.txt', 'r') as file:
    lines = file.readlines()
```

In summary:

- Use `read()` when you want to read the entire content of the file as a single string.

- Use `readline()` when you want to read the file line by line, or you only need to process one line at a time.

- Use `readlines()` when you want to read all the lines of the file into a list, especially if you need to iterate over them or access them randomly.

Writing to a File:

When you open a file in write mode, you can write data to it using the `write()` method.

```
# Opening a file in write mode
file = open("example.txt", "w")

# Writing data to the file
file.write("Hello, World!\n")
```

Appending to a File:

Appending to a file is similar to writing, but it doesn't erase the existing content. Instead, it adds new content to the end of the file.

```
# Opening a file in append mode
file = open("example.txt", "a")

# Appending data to the file
file.write("Appending new line!")
```

Closing a File:

It's essential to close the file once you're done working with it to free up system resources.

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

Context Managers (with Statement):

Python's `with` statement provides a more concise way to open and work with files. It automatically closes the file when the block inside `with` is exited.

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

Error Handling:

It's crucial to handle exceptions that might occur during file operations, such as `FileNotFoundError` or `PermissionError`.

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied!")
finally:
    file.close()
```

Difference between write() and writelines() :

In Python text file handling, `write()` and `writelines()` are methods used to write data into a file, but they differ in how they handle input and where they position the file cursor. Here's a breakdown of the differences:

1. **write()**:

- `write()` is a method used to write a string to a file.

- It writes the string directly to the file without adding any line breaks unless explicitly included in the string.

- If the file is opened in text mode (`'t'`), it expects a string as input. If opened in binary mode (`'b'`), it expects a bytes-like object.

- After writing, the file cursor moves to the end of the written content.

Example:

```
1.    with open("file.txt", "w") as file:
2.        file.write("Hello, world!\n")
```

2. **writelines()**:

- `writelines()` is used to write a sequence of strings to a file.

- It takes an iterable (such as a list of strings) as input and writes each element of the iterable to the file.

- It does not automatically add line breaks between strings, so if line breaks are needed, they must be included in the strings themselves.

- After writing, the file cursor moves to the end of the last string written.

Example:

```
1. with open("file.txt", "w") as file:
2.        lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
3.        file.writelines(lines)
```

In summary, `write()` is used to write a single string to a file, `writelines()` is used to write multiple strings from an iterable to a file.

Example program to use exception handling while writing content to a text file.

```
try:
    # Open the file in write mode
    with open("output.txt", "w") as file:
        # Write data to the file
        file.write("Hello, world!\n")
        file.write("This is a test.\n")
        file.write("Writing to a text file using Python.\n")
        # Simulate an error by attempting to write to a closed file
        # This will trigger an IOError, which we'll handle
        file.write("This will cause an error.\n")
except IOError as e:
    # Handle the IOError exception
    print("An error occurred:", e)
else:
    # Code inside else block executes if no exception occurs
    print("Data has been written to the file successfully.")
finally:
    # This block will always execute, regardless of whether an exceptio
    n occurred
    print("Closing the file.")
```