File IO basics

File Objects and Modes:

- **Opening Files:** The `open()` function is used to open a file for reading, writing, or appending. It takes two arguments:
  - `filename`: The name (or path) of the file you want to work with.
  - `mode`: A string indicating the access mode (read, write, append, etc.). Here are common modes:
    - `'r'`: Open the file for reading (default mode). Raises an error if the file doesn't exist.
    - `'w'`: Open the file for writing. Creates a new file if it doesn't exist. Overwrites existing content.
    - `'a'`: Open the file for appending. Creates a new file if it doesn't exist. Adds new content to the end of the existing content.
    - `'x'`: Open the file for exclusive creation. Creates a new file and raises an error if the file already exists.
    - `'b'`: Open the file in binary mode (for non-text files).
  - You can also combine these modes with `'+'` to allow both reading and writing (e.g., `'r+'`, `'a+'`).
- **File Object:** When you open a file, `open()` returns a file object, which represents the open file and provides methods for interacting with it.

2. Reading from Files:

- `read()` **method:** Reads the entire content of the file into a string. Use this for small files.
- `readline()` **method:** Reads a single line from the file and returns it as a string (including the newline character).
- `readlines()` **method:** Reads all lines from the file and returns them as a list of strings (each element is a line).
- **Iterating over lines:** You can directly iterate over lines in a file using a `for` loop:

Python

```python
with open('myfile.txt', 'r') as file:
  for line in file:
    print(line, end='')  # Optional: `end=''` to avoid extra newline
```

3. Writing to Files:

- `write()` **method:** Writes a string to the file. The string is appended to the existing content if the mode allows it.

4. Closing Files:

- It's crucial to close files using the `close()` method on the file object when you're done with them. This ensures proper resource management and prevents data corruption. Python also provides a `with` statement that automatically closes the file upon exiting the code block:

Python

```
with open('myfile.txt', 'w') as file:
  file.write("This is some text written to the file.")
```

5. Additional Considerations:

- **Binary vs. Text Files:** By default, Python handles files in text mode. If you're working with non-text data (like images), use binary mode (`'b'`) to preserve the exact byte representation.
- **Error Handling:** Use `try...except` blocks to handle potential errors during file operations, such as file not found exceptions or permission issues.

Remember:

- Always close files after use.
- Choose the appropriate mode based on your needs (reading, writing, appending).
- Consider binary mode for non-text files.
- Handle file-related errors gracefully.

There are two main ways to close a file in Python:

1. Using the `close()` method:

This is the most basic approach. After you're done working with the file object, you can explicitly call the `close()` method on it:

Python

```
# Open a file for writing
file = open("myfile.txt", "w")

# Write some content to the file
file.write("This is some text written to the file.")
```

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

2. Using the `with` statement:

The `with` statement provides a more convenient and safer way to handle file I/O in Python. It automatically closes the file object when the code block within the `with` statement exits, even if there are exceptions. Here's an example:

Python

```
# Open a file for writing with the 'with' statement
with open("myfile.txt", "w") as file:
  file.write("This is some text written to the file.")

# No need to call close() explicitly, the file is closed automatically

# You can access the file object within the 'with' block
print(file.closed)  # Output: True (file is closed)
```

Here's why using `with` is preferred:

- **Automatic closing:** Ensures the file is always closed, even if errors occur.
- **Cleaner code:** Eliminates the need for explicit `close()` calls, making the code more concise.
- **Exception safety:** If an exception happens within the `with` block, the file is still closed properly.

In summary:

For most cases, using the `with` statement is the recommended way to close files in Python. It simplifies your code and guarantees proper resource management. You can still use the `close()` method manually if needed, but the `with` statement often provides a more robust and convenient approach.

```

f = open("david.txt") // file pointer
content = f.read()
print(content)  # prints all the contents of the file

f.close()
```

File closing :

The need of file closing :
1. Resource Management:

- Opening a file allocates system resources, such as memory buffers and file handles. These resources are limited, and leaving files open unnecessarily can potentially deplete them.
- Closing a file releases these resources back to the system, making them available for other applications or processes. This is especially crucial for large files or when working with many files simultaneously.

2. Data Consistency:

- In some cases, changes made to a file might not be reflected on disk until the file is closed. This is because Python might buffer the data in memory before writing it permanently.
- Closing the file ensures that all pending writes are flushed from the buffers to the disk, guaranteeing data consistency. This is important to prevent data loss or corruption, especially when multiple processes might be accessing the same file.

3. Error Handling:

- If an error occurs while working with a file (e.g., disk full, permission issues), leaving the file open might prevent other processes from accessing it.
- Closing the file ensures it's properly released, even in case of exceptions, allowing other applications to interact with the file if necessary.

4. Program Efficiency:

- While not always a major factor, keeping too many files open can slightly impact performance. Closing files reduces the number of open file handles and frees up memory buffers, which can contribute to a more efficient program overall.

5. Good Programming Practice:

- Closing files is considered good programming practice in Python. It demonstrates responsible resource management and helps prevent potential issues down the line, especially in larger or more complex applications.

```
f = open("david.txt") // file pointer
content = f.read(3)
```

```
print(content)  # prints the first 3 characters of the file

content = f.read(3)
print(content)  # prints the next 3 characters of the file

f.close()
```

```


f= open("David.txt" , "rt")

for line in f :
        print (line , end ="") // the end dekhi ko code le chahi naya line
aaundaina
// this code prints all the lines of the file
```

```
f= open("David.txt" , "rt")
print(f.readline()) // prints the first line of the file
print(f.readline()) // prints the second line of the file
print(f.readline()) // prints the third line of the file
```

```
f= open("David.txt" , "rt")
print(f.readlines()) // stores and prints all the characters of the file in a
list
```
