EXPERIMENT NO.3

Title: Program for File Handling in Python.

Objective:

The aim of this experiment is to delve into the fundamentals of file handling in Python and analyze the performance of various file handling methods. Through this experiment, students will gain insights into the efficiency of file operations such as reading, writing, and appending data, and understand the impact of file size on I/O operations.

Theory:

Python File Handling

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, like other concepts of Python, this concept here is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters, and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let's start with the reading and writing files.

- **Advantages of File Handling in Python:**

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.
2. **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.).
3. **User–friendly**: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.

4. **Cross-platform**: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

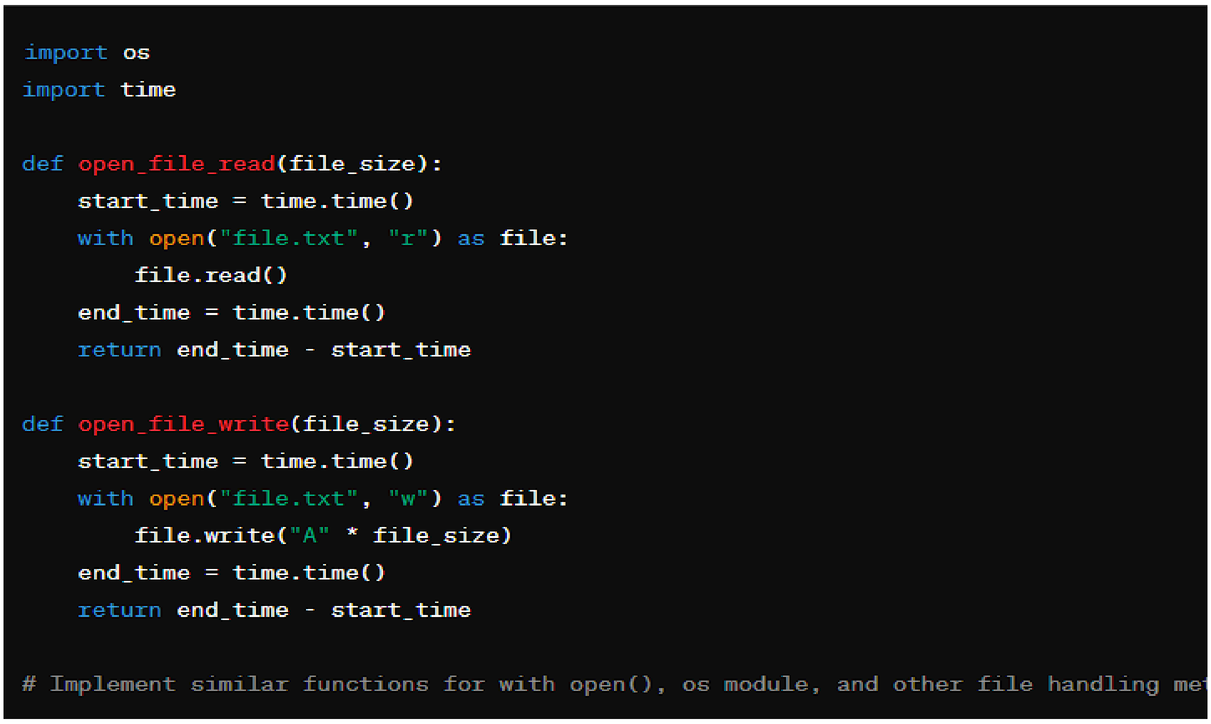
- **Disadvantages of File Handling in Python:**

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.).
2. **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.
3. **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.
4. **Performance:** File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.
5. File handling in Python involves various methods for reading from and writing to files, providing flexibility and control over data manipulation.

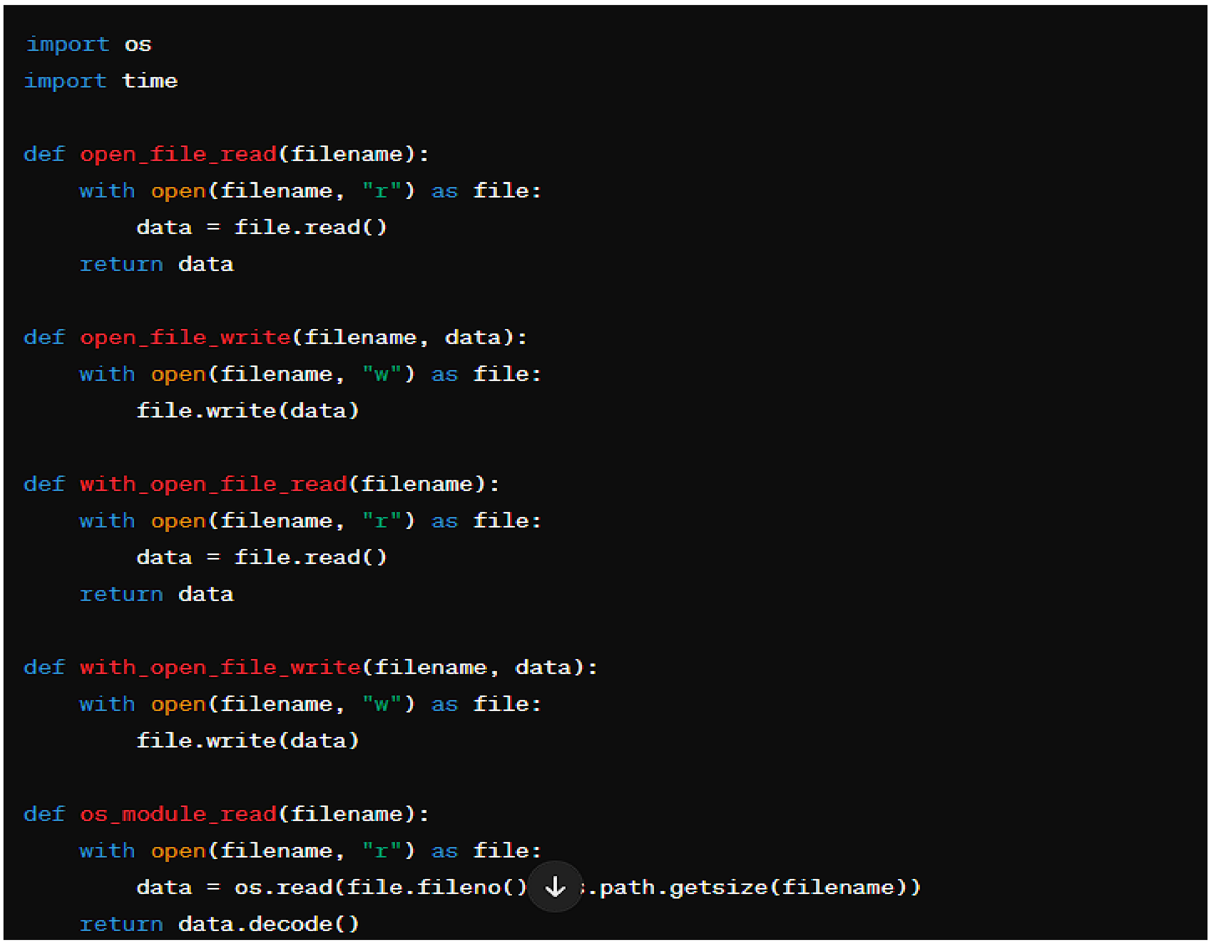
- **open() Function:** This built-in function serves as the primary method for file handling in Python. It allows the programmer to open files in different modes (read, write, append, etc.), returning a file object that can be manipulated to perform various operations.
- **with open() Statement:** The with statement in Python is a context manager that ensures proper resource management, especially concerning file handling. When combined with the open() function, it automatically closes the file once the block of code inside it is executed, preventing resource leaks and improving code readability.
- **os Module:** The os module provides a platform-independent interface to interact with the operating system, including file I/O operations. While higher-level functions like open() offer simplicity and ease of use, the os module provides lower-level access, allowing direct interaction with file descriptors and system calls.
- **Read and Write Methods:** Reading from and writing to files are fundamental file handling operations in Python.

- **Read:** Reading data from a file can be accomplished using methods like read(), readline(), or readlines(). These methods allow the programmer to retrieve data from the file either as a whole, line by line, or as a list of lines.
- **Write**: Writing data to a file involves methods like write() and writelines(). These methods enable the programmer to write data to the file, either as a single string or a list of strings.
- here's a simple Python code that includes file creation, reading, writing, and appending operations using different file handling methods:

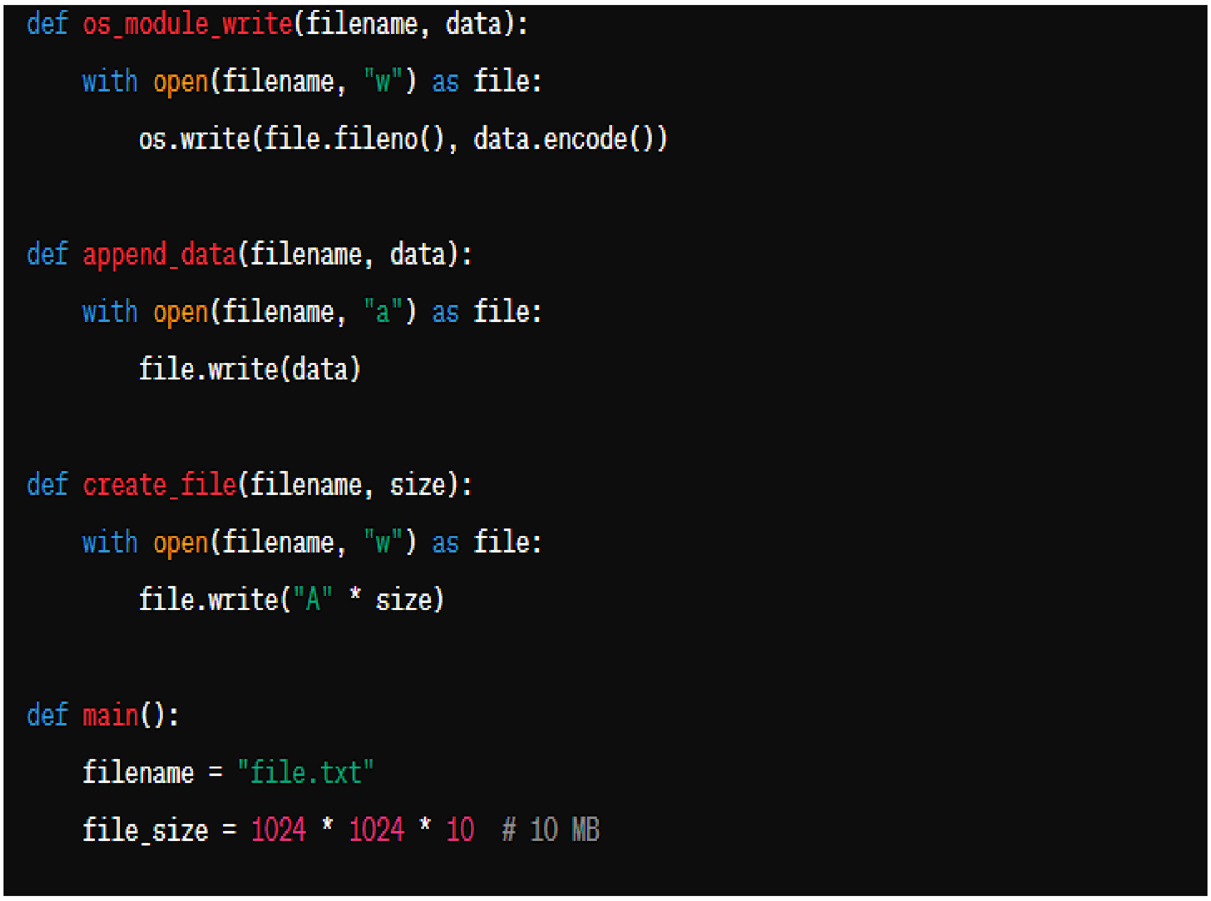
PROGRAM:



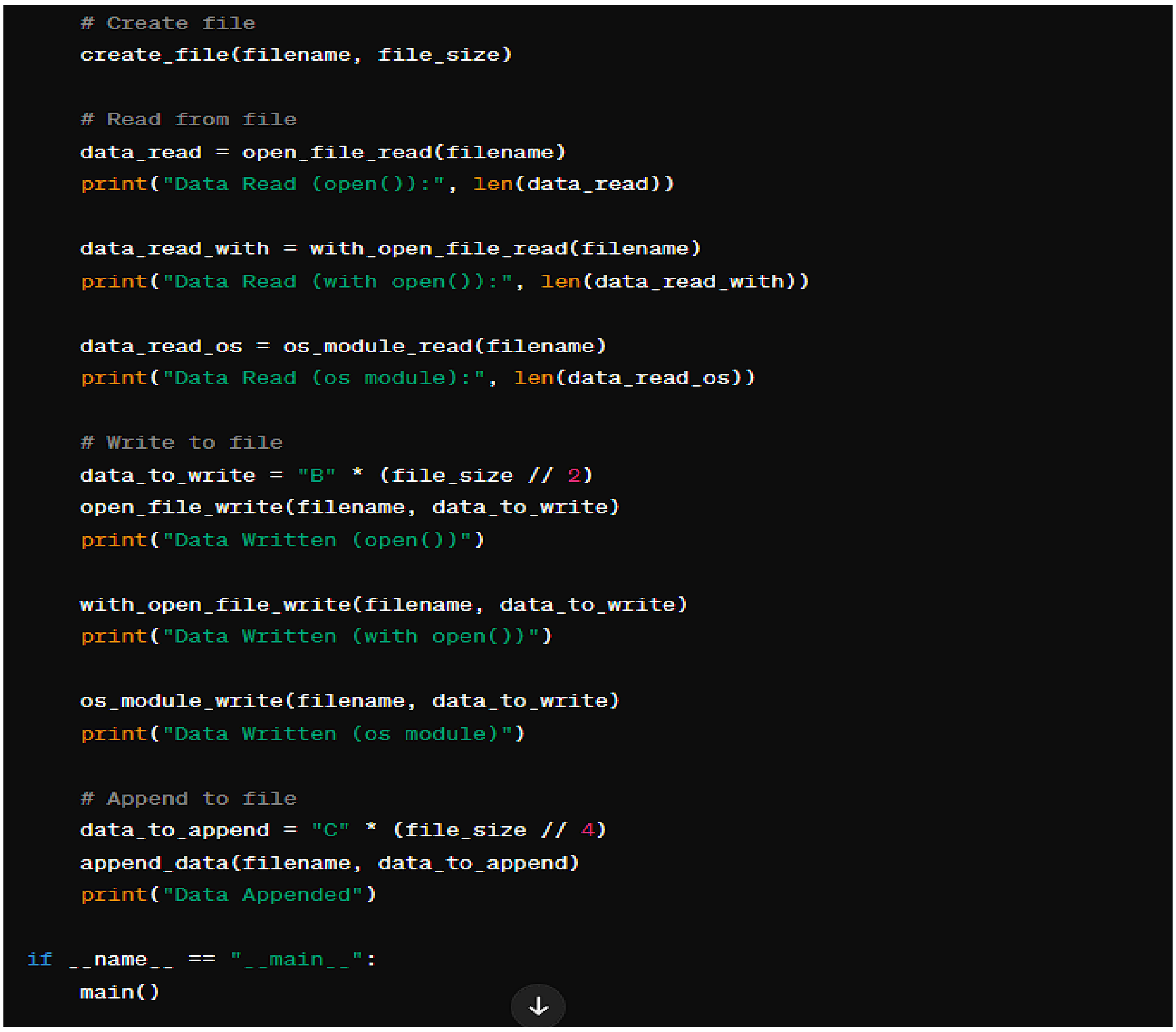
```python
import os
import time

def open_file_read(file_size):
    start_time = time.time()
    with open("file.txt", "r") as file:
        file.read()
    end_time = time.time()
    return end_time - start_time

def open_file_write(file_size):
    start_time = time.time()
    with open("file.txt", "w") as file:
        file.write("A" * file_size)
    end_time = time.time()
    return end_time - start_time

# Implement similar functions for with open(), os module, and other file handling met
```



```python
import os
import time

def open_file_read(filename):
    with open(filename, "r") as file:
        data = file.read()
    return data

def open_file_write(filename, data):
    with open(filename, "w") as file:
        file.write(data)

def with_open_file_read(filename):
    with open(filename, "r") as file:
        data = file.read()
    return data

def with_open_file_write(filename, data):
    with open(filename, "w") as file:
        file.write(data)

def os_module_read(filename):
    with open(filename, "r") as file:
        data = os.read(file.fileno() [illegible]s.path.getsize(filename))
    return data.decode()
```



```python
def os_module_write(filename, data):
    with open(filename, "w") as file:
        os.write(file.fileno(), data.encode())

def append_data(filename, data):
    with open(filename, "a") as file:
        file.write(data)

def create_file(filename, size):
    with open(filename, "w") as file:
        file.write("A" * size)

def main():
    filename = "file.txt"
    file_size = 1024 * 1024 * 10  # 10 MB
```



```
    # Create file
    create_file(filename, file_size)

    # Read from file
    data_read = open_file_read(filename)
    print("Data Read (open()):", len(data_read))

    data_read_with = with_open_file_read(filename)
    print("Data Read (with open()):", len(data_read_with))

    data_read_os = os_module_read(filename)
    print("Data Read (os module):", len(data_read_os))

    # Write to file
    data_to_write = "B" * (file_size // 2)
    open_file_write(filename, data_to_write)
    print("Data Written (open())")

    with_open_file_write(filename, data_to_write)
    print("Data Written (with open())")

    os_module_write(filename, data_to_write)
    print("Data Written (os module)")

    # Append to file
    data_to_append = "C" * (file_size // 4)
    append_data(filename, data_to_append)
    print("Data Appended")

if __name__ == "__main__":
    main()
```

- **Conclusion:**

1.File handling in Python offers multiple methods, each catering to specific requirements and preferences of programmers.

2.Understanding the nuances of reading from and writing to files is essential for efficient data manipulation and I/O operations in Python programs.

3.The experiment provides valuable insights into the performance characteristics of file handling methods, aiding in informed decision-making during application development.