File I/O Handling

File handling in Python allows for user-friendly operations to work with files. Through Python's built-in functions, one can open, read, write, and close files. This permits data storage and retrieval, making it possible to persistently work with large datasets and perform file-related tasks like data analytics, log parsing, and more. Python's file handling mechanisms are OS-independent, ensuring smooth operations across different platforms.

1. Writing to a file

In [1]:
```
with open('sample.txt', 'w') as file:
    file.write('Hello, World!')
```

2. Reading from a file

In [2]:
```
with open('sample.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
Hello, World!
```

In [3]:
```
# File Opening Modes
# 'r' - Default value. Opens a file for reading, error if the file doe
# 'a' - Opens a file for appending, creates the file if it does not ex
# 'w' - Opens a file for writing, creates the file if it does not exis
# 'x' - Opens a file for exclusive creation, if the file exists the op
```

3. Appending to a file

In [4]:
```
with open('sample.txt', 'a') as file:
    file.write('\nAppended Text!')
```

4. Trying exclusive creation mode

In [5]:
```
try:
    with open('sample.txt', 'x') as file:
        file.write('Exclusive Creation!')
except FileExistsError:
    print("File already exists!")
```

```
File already exists!
```

Manipulating files and directories

In [6]:
```
import os
```

5. Rename a file

In [7]:
```
os.rename('sample.txt', 'new_sample.txt')
```

6. Remove a file

In [8]:
```
os.remove('new_sample.txt')
```

7. Create a directory

In [9]:
```
os.mkdir('sample_directory')
```

8. Change current working directory

In [10]:
```
os.chdir('sample_directory')
```

9. Get current working directory

In [11]:
```
print(os.getcwd())  # Outputs the path of the 'sample_directory'
```

```
/Users/josephkambham/Downloads/sample_directory
```

10. Remove a directory

In [12]:
```
os.chdir('../')
os.rmdir('sample_directory')
```

Exception Handling

Exception handling is a mechanism in programming that handles runtime errors, ensuring that the execution of the program doesn't abruptly halt. In Python, this is achieved using the try, except blocks. The code that might raise an exception is placed inside the try block. If an error occurs, the code inside the except block is executed.

In [13]:
```
def divide_numbers(num1, num2):
    try:
        result = num1 / num2
        return result
    except ZeroDivisionError:
        return "Division by zero is not allowed!"
    except TypeError:
        return "Please enter numbers only!"
    finally:
        print("Function executed")

# Testing the function
print(divide_numbers(10, 2))  # Expected: 5.0
print(divide_numbers(10, 0))  # Expected: Division by zero is not allo
print(divide_numbers(10, 'a'))  # Expected: Please enter numbers only!
```

```
Function executed
5.0
Function executed
Division by zero is not allowed!
Function executed
Please enter numbers only!
```

In [14]:
```
# In this example, we have a function that divides two numbers.
```

In [ ]:
```
```