# Memory File System

### Class: InMemoryFileSystem

class InMemoryFileSystem:

def \_\_init\_\_(self):

self.current\_directory = '/'

self.file\_system = {}

The **InMemoryFileSystem** class is initialized with the root directory **'/'** and an empty file system represented by a dictionary.

### Method: mkdir(self, directory)

def mkdir(self, directory):

path = self.\_get\_absolute\_path(directory)

if path not in self.file\_system:

self.file\_system[path] = {}

else:

print(f"Directory '{directory}' already exists.")

Creates a new directory. If the directory already exists, it prints a message.

### Method: cd(self, path)

def cd(self, path):

if path == '/':

self.current\_directory = '/'

else:

new\_path = os.path.normpath(os.path.join(self.current\_directory, path))

if new\_path in self.file\_system and isinstance(self.file\_system[new\_path], dict):

self.current\_directory = new\_path

else:

print(f"Invalid path: '{path}'")

* Changes the current directory. If the specified path is valid, it updates the current directory; otherwise, it prints an error message.

**Method: ls(self, directory=None)**

def ls(self, directory=None):

if directory is None:

directory = self.current\_directory

if directory in self.file\_system and isinstance(self.file\_system[directory], dict):

contents = list(self.file\_system[directory].keys())

print("\n".join(contents))

else:

print(f"Invalid directory: '{directory}'")

* Lists the contents of a directory. If no directory is specified, it lists the contents of the current directory.

**Method: touch(self, filename)**

def touch(self, filename):

path = os.path.join(self.current\_directory, filename)

if path not in self.file\_system:

self.file\_system[path] = ""

else:

print(f"File '{filename}' already exists.")

* Creates a new empty file. If the file already exists, it prints a message.

**Method: echo(self, filename, content)**

def echo(self, filename, content):

path = os.path.join(self.current\_directory, filename)

if path in self.file\_system:

self.file\_system[path] = content

else:

print(f"File '{filename}' does not exist.")

* Writes content to a file. If the file exists, it updates the content; otherwise, it prints an error message.

**Method: cat(self, filename)**

def cat(self, filename):

path = os.path.join(self.current\_directory, filename)

if path in self.file\_system:

print(self.file\_system[path])

else:

print(f"File '{filename}' not found.")

* Reads and prints the content of a file. If the file doesn't exist, it prints an error message.

**Method: mv(self, source, destination)**

def mv(self, source, destination):

src\_path = os.path.join(self.current\_directory, source)

dest\_path = os.path.join(self.current\_directory, destination)

if src\_path in self.file\_system:

self.file\_system[dest\_path] = self.file\_system.pop(src\_path)

else:

print(f"Source file '{source}' not found.")

* Moves or renames a file or directory. If the source doesn't exist, it prints an error message.

**Method: cp(self, source, destination)**

def cp(self, source, destination):

src\_path = os.path.join(self.current\_directory, source)

dest\_path = os.path.join(self.current\_directory, destination)

if src\_path in self.file\_system:

self.file\_system[dest\_path] = self.file\_system[src\_path]

else:

print(f"Source file '{source}' not found.")

* Copies a file or directory. If the source doesn't exist, it prints an error message.

**Method: rm(self, path)**

def rm(self, path):

full\_path = os.path.join(self.current\_directory, path)

if full\_path in self.file\_system:

if isinstance(self.file\_system[full\_path], dict) and self.file\_system[full\_path]:

print(f"Cannot remove non-empty directory '{path}'.")

else:

self.file\_system.pop(full\_path)

else:

print(f"File or directory '{path}' not found.")

* Removes a file or an empty directory. If the path is a non-empty directory, it prints an error message.

**Method: \_get\_absolute\_path(self, path)**

def \_get\_absolute\_path(self, path):

if path.startswith('/'):

return path

* Returns the absolute path for a given path, handling relative paths.

**Method: save\_state(self, save\_path)**

def save\_state(self, save\_path):

with open(save\_path, 'w') as file:

json.dump({'current\_directory': self.current\_directory, 'file\_system': self.file\_system}, file)

* Saves the current state of the file system to a JSON file.

**Method: load\_state(self, load\_path)**

def load\_state(self, load\_path):

with open(load\_path, 'r') as file:

data = json.load(file)

self.current\_directory = data['current\_directory']

self.file\_system = data['file\_system']

* Loads the file system state from a JSON file.

**Main Block:**

if \_\_name\_\_ == "\_\_main\_\_":

file\_system = InMemoryFileSystem()

while True:

command = input(f"{file\_system.current\_directory} $ ").split()

if command:

operation = command[0]

if operation == 'exit':

break

elif operation == 'save\_state':

file\_system.save\_state(command[1])

elif operation == 'load\_state':

file\_system.load\_state(command[1])

else:

try:

getattr(file\_system, operation)(\*command[1:])

except AttributeError:

print(f"Invalid command: '{operation}'")

print("Exiting the file system.")

The main block runs a loop to accept user commands for interacting with the in-memory file system. The user can perform various operations like creating directories, navigating, creating files, etc. The loop continues until the user enters the 'exit' command. The program also supports saving and loading the file system state.