Section 9: File Handling and OS Operations

This section introduces you to file handling in Python, which allows your programs to interact with files on your computer. We'll also explore basic operating system (OS) interactions using Python's built-in modules.

File I/O in Python

File Input/Output (I/O) refers to reading data from and writing data to files. Python provides built-in functions to make this process straightforward. Working with files generally involves these steps:

1. **Opening a file:** You need to open a file before you can read from it or write to it. This creates a connection between your program and the file.
2. **Performing operations:** You can then read data from the file or write data to it.
3. **Closing the file:** It's crucial to close the file when you're finished with it. This releases the connection and ensures that any changes you've made are saved.

Read, Write, and Append Files

Python provides several modes for opening files:

- **'r' (Read mode):** Opens the file for reading. This is the default mode. If the file doesn't exist, you'll get an error.
- **'w' (Write mode):** Opens the file for writing. If the file exists, its contents will be overwritten. If the file doesn't exist, a new file will be created.
- **'a' (Append mode):** Opens the file for appending. Data will be added to the end of the file. If the file doesn't exist, a new file will be created.

Here are some examples:

Reading from a file:

```
try:
    file = open("my_file.txt", "r")  # Open in read mode
    content = file.read()  # Read the entire file content
    print(content)
    file.close()  # Close the file
except FileNotFoundError:
    print("File not found.")


# Reading line by line
try:
    file = open("my_file.txt", "r")
    for line in file: # Efficient for large files
        print(line.strip()) # Remove newline characters
    file.close()
except FileNotFoundError:
    print("File not found.")
```

Writing to a file:

```
file = open("new_file.txt", "w")  # Open in write mode (creates o
file.write("Hello, world!\n")  # Write some text
file.write("This is a new line.\n")
file.close()
```

Appending to a file:

```
file = open("my_file.txt", "a")  # Open in append mode
file.write("This is appended text.\n")
file.close()
```

Using `with` statement (recommended):

The `with` statement provides a cleaner way to work with files. It automatically closes the file, even if errors occur.

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

```python
except FileNotFoundError:
    print("File not found.")

with open("output.txt", "w") as file:
    file.write("Data written using 'with'.\n")
```

OS and Shutil Modules in Python

Python’s `os` module provides functions for interacting with the operating system, such as working with directories and files. The `shutil` module offers higher-level file operations.

`os` module examples:

```python
import os

# Get the current working directory
current_dir = os.getcwd()
print("Current directory:", current_dir)

# Create a new directory
# os.mkdir("new_directory")  # creates only one level of director
# os.makedirs("path/to/new_directory") # creates nested directori

# Change the current directory
# os.chdir("new_directory")

# List files and directories in a directory
files = os.listdir(".") # "." represents current directory
print("Files in current directory:", files)

# Remove a file or directory
# os.remove("my_file.txt")
# os.rmdir("new_directory") # removes empty directory
# shutil.rmtree("path/to/new_directory") # removes non-empty dire

# Rename a file or directory
# os.rename("old_name.txt", "new_name.txt")
# Check if a file or directory exists
if os.path.exists("my_file.txt"):
    print("File exists")

# Join path components in a platform-independent way
path = os.path.join("folder", "subfolder", "file.txt")
print("Joined path:", path)
```

`shutil` module examples:

```python
import shutil

# Copy a file
# shutil.copy("my_file.txt", "my_file_copy.txt")

# Move a file or directory
# shutil.move("my_file.txt", "new_directory/")
```

Creating Command Line Utilities

You can use Python to create simple command-line utilities. The `argparse` module makes it easier to handle command-line arguments.

```python
import argparse

parser = argparse.ArgumentParser(description="A simple command-li
parser.add_argument("filename", help="The file to process.")
parser.add_argument("-n", "--number", type=int, default=1, help="

args = parser.parse_args()

try:
    with open(args.filename, "r") as file:
        content = file.read()
        for _ in range(args.number):
            print(content)
```

```
except FileNotFoundError:
    print("File not found.")
```

To run this script from the command line:

```
python my_script.py my_file.txt -n 3
```

This will print the contents of `my_file.txt` three times. You can learn more about `argparse` in the Python documentation.