SDEV 1001

Programming Fundamentals

Dictionaries - 1

A LEADING POLYTECHNIC COMMITTED TO YOUR SUCCESS

Expectations - What I expect from you

- No Late Assignments

- No Cheating

- Be a good classmate

- Don’t waste your time

- Show up to class

Agenda

On the right is what we will cover today.

- Reading and Writing Files in Python
- Reading a File
- Handling Missing Files
- Writing to a File
- User Interaction Example
- Best Practices

Reading and Writing Files in Python

Why is this important?

- File I/O (Input/Output) is essential for data persistence.
- Allows programs to save and load data between runs.
- Commonly used for logs, user data, and configuration files.
- Although some libraries you'll learn from later will abstract this away, it's important to understand the fundamentals.

What is File I/O?

- Files let you store and retrieve data between program runs.
- Common use cases: saving user data, logs, or configuration.

Reading a File

To read a file in Python, you can use the `open()` function with the `'r'` mode. Here's a simple example:

The `f.readlines()` method reads all lines in a file and returns them as a list. There's also `f.read()` to read the entire file as a single string, and `f.readline()` to read one line at a time.

- Note `f.readline()` reads one line at a time, which is useful for large files (you'll just use a loop to read each line).

```
def read_file(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
    return lines

for line in read_file("notes.txt"):
    print(line.strip())
```

Handling Missing Files

To handle cases where a file might not exist, you can use `try` / `except` blocks. This allows your program to continue running even if the file is missing.

```
def safe_read(filename):
    try:
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        return "File not found."

print(safe_read("missing.txt"))
```

Writing to a File

Writing to a file is done using the `open()` function with the `'w'` or `'a'` mode. The `'w'` mode overwrites the file, while `'a'` appends to it.

- Each write adds a new line.

```
def write_file(filename, text):
    with open(filename, 'a') as f:
        f.write(text + "\n")

write_file("notes.txt", "Remember to review Python file I/O.")
```

User Interaction Example

We’re going to do something similar to this application but with a simple to-do list application. This will allow users to add items to a list and read them from a file.

- Now if you run this code several times, it will keep adding items to the file without overwriting previous entries.
- It will also persist the data even after the program exits.

```
filename = "todo.txt"
while True:
    action = input("Add (a), Read (r), or Quit (q)? ").lower()
    if action == "a":
        item = input("Enter a to-do item: ")
        write_file(filename, item)
    elif action == "r":
        for line in read_file(filename):
            print("-", line.strip())
    elif action == "q":
        break
```

- Simple menu for reading and writing to a file.

Best Practices

- Always close files (use `with` to do this automatically).
- Handle exceptions for missing or locked files.
- Use clear file paths and check for file existence if needed.

Summary

- Reading and writing files is essential for persistent data.
- Use `open()`, `read()`, `readlines()`, and `write()` for file operations.
- Handle errors gracefully and always close your files.

Example

Let’s go run a few examples together