ITEC 1150 Week 12
Chapter 9
READING AND WRITING FILES

File Systems

IN THIS WEEK'S LAB AND HOMEWORK, WE WORK WITH READING AND WRITING TO TEXT FILES IN OUR PYTHON PROGRAMS.

BEFORE WORKING WITH ANY FILE TYPE, WE NEED TO BE COMFORTABLE WITH THE CONCEPTS, AND WITH PYTHON METHODS TO INTERACT WITH THE COMPUTER'S FILE SYSTEM.

File Systems Basic Concepts

- Windows and MacOS operating systems try to hide their file systems to an extent, and mobile OSes hide them go a much greater extent, but they all work the same under the hood.
- On every operating system, there are **directories** (which non-programmers call *folders*), which can contain **files** and/or other directories.
- Files are things like applications, documents, videos, etc. These are what take up space on your hard drive or device storage. (Directories don't take up any space, but the files in them do.)
- There's always one directory called the **root**, which isn't in any other directory—it's at the top of the file system, and every other directory or file can be identified based on where it is relative to the root.
- (On Windows, the root is usually named **`c:`** and on MacOS and Unix, it's always named **`/`**)

- The first level of directories in a typical Windows installation:

```
c:
├── $Recycle.Bin
├── Config.Msi
├── Documents and Settings
├── Drivers
├── PerfLogs
├── Program Files
├── Program Files (x86)
├── ProgramData
├── Recovery
├── System Volume Information
├── Users
├── Windows
└── XboxGames
```

Paths

- A **path** identifies a file or a directory.
- In Windows, parts of the path are separated with backslashes ("\"):

```
C:\users\username\PycharmProjects\color_mixer.py
```

- In MacOS and Linux, parts of the path are separated with slashes ("/"):

```
/users/username/PycharmProjects/color_mixer.py
```

- In both examples on the left,
  - the file `color_mixer.py` is in a directory named
  - `PycharmProjects`, which is inside a directory named
  - `username`, which is inside a directory named
  - `users`, which is inside
  - the root directory.
- These are the **full path**s for `color_mixer.py`.

Full and Relative Paths

- A path can either be absolute, or relative:
  - An absolute path starts at the root of the file system and contains the full path
  - A relative path is relative to some directory and does not contain the full path.
- Most operating systems use special characters to help navigate relative paths:
  - A single dot means "the current directory"
  - Two dots means "one level up"
- If my program is in the directory `C:\users\username\` then the relative path to `color_mixer.py` is `.\PycharmProjects\color_mixer.py`
  - Notice the single dot in the line above!

Working With Paths in Python

Program Portability

- **Portability** refers to writing a program so that anyone else can run it, regardless of things like their operating system.
- Because different operating systems use different characters for defining paths, simple string concatenation is a bad idea for building paths.
  - For example, if you use "\" as the path separator because you're on Windows, if someone runs your program on an OS other than Windows, it will not work (and vice versa). This means your program is not portable.
- Python provides functions for building paths that figure out the correct path separator ("/", "\", or anything else) for your operating system so your program will work on any operating system.

Joining Paths

- The `pathlib` module is the most modern way of interacting with paths in Python. Run the following code to see what it outputs:

```
1. from pathlib import Path
2. print('Using the pathlib module and the Path function')
3. print("Results of running: Path('users', 'shared', 'android')")
4. path = Path('users', 'shared', 'android')
5. print(path)
```

- Note that just because you made a path in Python, it doesn't necessarily mean that path exists on your computer's file system!

Joining Paths (older method)

- If you search online for help with Python file operations, you will probably find a lot of references to an older module, `os.path`, which looks like this:

```
1. import os
2. print('Using the os module and the os.path module')
3. print("Results of running: os.path.join('users', 'shared', 'android')")
4. path = os.path.join('users', 'shared', 'android')
5. print(path)
```

- The `path` from the `pathlib` module is generally more robust than the `path` from the `os` module, although either will work.

Finding The Current Working Directory

- A program is always running in a directory. Unless you change it in your program, it will (almost always) be the directory in which the program is saved.
- Run the program below:

```
1. import os
2. print('Current working directory - compare yours to the example.')
3. print('Results of running: os.getcwd()') #CWD is short for 'current working directory'
4. curr_dir = os.getcwd()
5. print(curr_dir)
```

- One example of output is below; how does yours differ?

```
Results of running: os.getcwd()
/Users/erikgranse/src/ITEC_1150
```

Changing The Working Directory

```
1. import os
2. base_dir = os.getcwd()
3. print('Initial working directory: {}'.format(base_dir))
4. os.chdir(r'..') # Change to the directory above
5. print('One directory up: {}'.format(os.getcwd()))
6. os.chdir(base_dir)
7. print('And back to the initial directory: {}'.format(os.getcwd()))
```

Create A Precise Path To A File

- Let's use a loop and a list of file names to create multiple paths at once:

```
1. from pathlib import Path
2. print('Results of using os.path.join with a path and a list of file names')
3. my_files=['accounts.txt', 'details.csv', 'friends.docx']
4. for index in range(len(my_files)):
5.     print(Path('users', 'username', 'PycharmProjects', 'projectname', my_files[index]))
```

- Again, use with caution: you can create the paths even if they don't exist!
- What would happen if you told Python to open the file `/users/username/PycharmProjects/projectname/accounts.txt`?

File And Directory Errors

- Run this to see what happens if you try to change to a directory that doesn't exist:

```
1. import platform, os
2. if platform.system() == 'Windows':
3.     os.chdir(r'C:\Users\NoSuchDirectory')  # Windows
4. else:
5.     os.chdir('/Users/NoSuchDirectory') # OSX or Linux
```

- To avoid having an error stop your program, add exception handling:

```
1. import platform, os
2. try:
3.     if platform.system() == 'Windows':
4.         os.chdir(r'C:\Users\NoSuchDirectory')  # Windows
5.     else:
6.         os.chdir('/Users/NoSuchDirectory') # OSX or Linux
7. except FileNotFoundError:
8.     print("The file or directory does not exist!")
```

Creating New Directories

- The `os.makedirs()` function is used to make a new directory.
- You can only make a given directory once, so re-running this will give an error (unless you add exception handling!)
- The new directory will be created in your current working directory.

```
1. import os
2. from pathlib import Path
3. print('Creating new directory folders.')
4. my_path = os.getcwd() # save current working directory under variable name
5. print(my_path)
6. print('Create a new directory: test_dir')
7. os.makedirs(Path(my_path, 'test_dir')) # create test_dir under current directory
8. print('OK I think that worked – check directory to see new folder.')
9. print('Running again will cause an error if test_dir exists...')
```

Listing Directory Contents

- Print a list of the files and directories in the current working directory using `os.listdir()`. You can also check to see if something is a directory or is a file with `os.path.isdir()`:

```
1. import os
2. print('The files and directories in the current directory:')
3. dir_contents = os.listdir(os.getcwd())
4. for element in dir_contents:
5.     if os.path.isdir(element):
6.         print(f"{element} is a directory")
7.     else:
8.         print(f"{element} is a file")
```

Reading And Writing Files

NOW THAT WE CAN NAVIGATE OUR FILE STRUCTURE, IT'S TIME TO LEARN HOW TO READ AND WRITE THE CONTENTS OF FILES.

Opening And Closing Files

- To open a file, use the following command:
  - `my_file = open(file_name, mode)`
- The file name parameter can be a full path, a relative path, or just a file name (in which case Python will use the current working directory).
- The mode parameter accepts the following values:
  - `'w'`: Write. Opening a file with this mode will overwrite any contents already in the file.
  - `'a'`: Append. Opening a file with this mode will write any new content to the end of the file.
  - `'r'`: Read. No changes can be made to the file in this mode.
- When you are done with a file, you **must** call `close()` on it. If a file is not properly closed, the operating system might think it's still being used and not let programs read from or write to the file.
- We won't provide a path to a specific directory for now, so new files will appear in your current working directory

To Create a Text File

- Open a file object in 'w' mode to create the file.
- Run this code once, then predict the answers to the questions below.

```
1. my_file = open('my_file.txt', 'w') # w is for for write mode
2. my_file.write('Hello Earth!\n')
3. my_file.write('Hello Mars!\n')
4. my_file.close()
5. print('Done!')
```

- It says 'Done!' but what did it do? Look in your project file list and double click on `my_file.txt`. As you can see, opening a file and writing to it creates the file.
  - What happens if you open and close it without writing anything?
  - What would happen if your program closed the file and re-opened it between the two write lines?

- File contents:

```
Hello Earth!
Hello Mars!
```

Append To A File

- Once a file exists, you should only use the 'w' mode if you want to overwrite the file (meaning erasing everything that's already in the file!).
- Let's try opening the file in append mode.
- Try this now, without re-running the previous code:

```
1. my_file = open('my_file.txt', 'a') # 'a' is for append mode
2. my_file.write('Hello Venus!\n')
3. my_file.write('Hello Jupiter!\n')
4. my_file.close()
5. print('Done again!')
```

- Note that you can open a file in append mode even if it doesn't already exist. Append mode is safest unless you are sure you want to overwrite the contents of any existing file.

- File contents:

```
Hello Earth!

Hello Mars!

Hello Venus!

Hello Jupiter!
```

Reading An Entire File

- In real-life situations, your program may need to use the data that's been saved into a file, so let's read the file through your program.
- Open the file in 'r' mode and `read()` it:

```
1. my_file = open('my_file.txt', 'r') # r is for read mode
2. my_data = my_file.read() # read the file into one string
3. print(my_data)
4. my_file.close() # close the file
```

- Output:

```
Hello Earth!
Hello Mars!
Hello Venus!
Hello Jupiter!
```

Reading A Text File Into A List

- We can also read the entire file into a list so we can process it further.
- For the `file.readlines()` method to work, each line must end with a `\n` newline character

```
1. my_file = open('my_file.txt', 'r') # r for read mode
2. my_data = my_file.readlines() # read the file into a list of lines
3. print(my_data)
4. my_file.close() # close the file
```

- Of course, we could loop through each line in the list just like any other list.

Read A Single line - and loop to the next

- With large files, reading the whole file in at once with either `read()` or `readlines()` can cause memory problems. Instead, the `readline()` method will read lines from a file, one at a time.

```
1. my_file = open('my_file.txt', 'r') # r for read mode
2. my_line = my_file.readline() # read the first line
3. print('First line from file: ' + my_line)
4. # here it's used with a loop
5. while my_file != '':
6.     print(my_line, end='') # the string from the file has a newline already,
7.                            so we don't want print() to add one
8.     my_line = my_file.readline() # loop through a line at a time
9. my_file.close() # close the file
```

If we didn't read the first line here, what would the loop print?

An Alternative Way Of Opening Files

- Because it's easy to forget to close a file, there's another way to open a file that will automatically close the file:

```
1. with open('my_file.txt', 'r') as my_file:
2.     for line in my_file:
3.         print(line, end='')
```

- This code is much simpler, but a word of warning—the file will be closed as soon as the block starting "with open" ends, so all your code interacting with the file must be indented into this block!
- If that becomes too complicated, use the methods we looked at in the earlier slides.

Putting It All Together – Sample Program

```
1.  PARTY_FILE = 'party_invites.txt'
2.  import os
3.  if os.path.isfile(PARTY_FILE) is False: # Only write the hostess if this is a new file
4.      invite_file = open(PARTY_FILE,'w')
5.      invite_file.write('Hostess - Betty Boop\n')
6.      invite_file.close()
7.  user_input = input('Enter guest names, separating each one with a comma. ')
8.  user_input_list = user_input.split(',') # change to a list
9.  invite_file = open(PARTY_FILE, 'a') # 'a' for append mode
10. for i in range(len(user_input_list)):
11.     guest_fixed = user_input_list[i].strip()
12.     user_input_fixed = 'Guest ' + str(i + 1) + ' - ' + guest_fixed + '\n' # added newline!
13.     invite_file.write(user_input_fixed)
14. invite_file.close()
```

Sample Program – Alternate

```
1.  PARTY_FILE = 'party_invites.txt'
2.  import os
3.  if os.path.isfile(PARTY_FILE) is False: # Only write the hostess if this is a new file
4.      with open(PARTY_FILE,'w') as invite_file:
5.          invite_file.write('Hostess - Betty Boop\n')
6.  user_input = input('Enter guest names, separating each one with a comma. ')
7.  user_input_list = user_input.split(',') # change to a list
8.  with open(PARTY_FILE, 'a') as invite_file: # 'a' for append mode
9.      for i in range(len(user_input_list)):
10.         guest_fixed = user_input_list[i].strip()
11.         user_input_fixed = 'Guest ' + str(i + 1) + ' - ' + guest_fixed + '\n' # added newline!
12.         invite_file.write(user_input_fixed)
```