System: Files

CPSC 217: Introduction to Computer Science for Multidisciplinary Studies I
July 2024

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

June 4, 2024

UNIVERSITY OF CALGARY

Files

- User input comes in various ways:
  - **Hard-code:** x = 10
  - **User keyboard input:** `info = input("Please enter the info")`
  - **Command-line arguments:** `>python programName arg1 ...`
  - **`File Input`**

- An input may be a path to a file (or to a directory where the file is located) that your program can process by:
  - Opening the file, reading from the file, overwriting or appending to the file, and closing the file.

Files

- There are two main file types:
  1. Text files
     - Encoded using ASCII or Unicode
     - Can be viewed with editors such as Emacs and Notepad
     - Examples: Python source files, web pages, …
  2. Binary files
     - Contain arbitrary sequences of bits which do not conform to ASCII or Unicode characters
     - Examples: Most images, word processor files, …

Paths

Files - File Path

- For a program to be able to access a file, you must tell it exactly where to find it by providing the **path** to the file.
- Example:

This is an **Absolute Path**, a path that starts from the root directory of your machine to a target file or directory.

Command-line:

```
C:\Users\kzaamout\Desktop>python test.py C:\Users\kzaamout\Desktop\input.csv
```

- Examples of Absolute Paths:
  - **Windows:** C:/users/Zaamout/Desktop/inputFile.csv
  - **Mac:** /users/Hudson/Desktop/inputFile.csv
  - **Linux (CPSC machines):** /home/grads/kzaamout/inputFile.csv

Files - Check if File Exists?

- You can check if a given file exists in Python using *path.exist()* function from the *os* library:
- The function returns a boolean.

Command-line:

```
C:\...\Desktop>python test.py test.py
exists? True

C:\...\Desktop>python test.py NOPE.txt
exists? False
```

test.py

```
import os
import sys

print("exist?", os.path.exists(sys.argv[1]))
```

- Notice that the files above did not have an absolute path. Why?

Files - File Path

- If the input file resides in the same location **from where you executed a module**, you can omit the path and only provide the input file name.
- Python searches for the file in the current working directory (this is where you executed the file.

Command-line:

```
C:\...\Desktop>python test.py test.py
exists? True

C:\...\kzaamout>python test.py test.py
exists? False
```

test.py

```
import os
import sys

print("exist?", os.path.exists(sys.argv[1]))
```

test.py is under Desktop directory, not kzaamout

HOW CAN YOU FIX THIS?

Open a File

Files - Opening a File

- Call Python's built-in **open()** function:

```
fileHandler = open(filePath, mode)
```

- **mode** can be: **'r'** for reading, **'w'** for writing (truncates the file first), **'a'** for appending (write to the end of the file), the default is **'r'**.

```
import sys
if len(sys.argv) != 2:
      print('missing file path‘)
else:
      filename = sys.argv[1]
      inputFileHandler = open(filename, "r")
```

Reading from a File

Files - Reading From a File

- Various ways to read files (in text mode):
  - data = inputFile.**read()** → Reads the entire file as one big string.
  - data = inputFile.**read(128)** → Reads 128 characters from the file.
  - data = inputFile.**readline()** → Reading one line from a file.
  - data = inputFile.**readlines()** → Read all lines from a file. The lines will be stored in a list and each line of the file is a list item.

Files - Reading From a File - For Loop Example

- This will print every line in the file named *inputFile.txt:*

```
filePath = 'C:/…/inputFile.txt'

fileHandler = open(filePath)
for line in fileHandler:
     print(line.rstrip())
```

Files - Reading From a File - For Loop Example

- This will print every line in the file named *inputFile.txt:*
- Alternatively, you can do the following:

```
filePath = 'C:/…/inputFile.txt'

lines = open(filePath).readlines()
for line in lines:
     print(line.rstrip())
```

Files - Reading From a File - While Loop Example

- This will also print every line in the file named *inputFile.txt:*

```
filePath = 'C:/…/inputFile.txt'

fileHandler = open(filePath)
line = fileHandler.readline()
while line != "":
     print(line.rstrip())
     line = fileHandler.readline()
```

Files - Reading From a File - File Pointer

- When open() is called, a file handler that contains a pointer is returned to the caller.

```
fileHandler = open(filePath))
```

- This pointer, initially, “points” at the beginning of the opened file.

```
print('Current position: %d’ % fileHandler.tell()) →
```

Files - Reading From a File - File Pointer

- When open() is called, a file handler that contains a pointer is returned to the caller.

```
fileHandler = open(filePath))
```

- As the program makes calls to various read functions, the pointer's position increments by the number of characters that have been read.

```
print('read: %s' % fileHandler.read(1))
```
→ char

```
print('Current position: %d' % fileHandler.tell())
```
→ 1

Files - Reading From a File - File Pointer

- You can use the *seek(offset, whence)* function to reposition the file pointer any where in the file:
  - **Offset** is the position to move to
  - **Whence** is an optional parameter that can be:
    - 0: the offset is an absolute position starting from the beginning of file.
    - 1: the offset is a relative position to the current pointer position
    - 2: the offset is from the end of the file.

```
fileHandler.seek(0, 0)
```

```
print('read: %s' % fileHandler.read(1))
```
→ prints the first character in the file.

Files - Reading From a File - File Pointer - Example

- Consider the file “inputFile.txt”. Each row consists of a 2 character long student ID, and a 4-character long student name.
- Retrieve the ID and name of the third student.
- One solution is to use loops.

inputFile.txt

```
ID NAME
01 Jack
02 Mack
03 Pham
...
```

Files - Reading From a File - File Pointer - Example

Program.py

```
filePath = 'C:/…/inputFile.txt'
fileHandler = open(filePath)

while counter <= 2:#skip the first three lines
    fileHandler.readline()#skip
    counter+=1
s_id = fileHandler.read(2)
fileHandler.read(1) #skip space
s_name = fileHandler.read(4)
print("Student ID: %s, Name: %s." % (s_id, s_name))
```

inputFile.txt

```
ID NAME
01 Jack
02 Mack
03 Pham
…
```

Console output:

```
Student ID: 03, Name: Pham.
```

Files - Reading From a File - File Pointer - Example

- Another solution is to use the file pointer directly.
- We know each row contains 7 characters:
  - 2 for ID
  - 1 space
  - 4 for Name
  - 1-2 newline characters (depending on operating system and the way you opened the file). In windows, I will have 2 newline characters at the end of each file.
  - 2+1+4+2=9 per row.
- We also know, we want the 3rd student; since we have a header row that makes him in the 4th row.

inputFile.txt

```
ID NAME
01 Jack
02 Mack
03 Pham
...
```

Files - Reading From a File - File Pointer - Example

Program.py

```
filePath = 'C:/…/inputFile.txt'
fileHandler = open(filePath)


fileHandler.seek(9*3) #skip the first 3 rows
s_id = fileHandler.read(2)
fileHandler.read(1) #skip space
s_name = fileHandler.read(4)


print("Student ID: %s, Name: %s." % (s_id, s_name))
```

inputFile.txt

```
ID NAME
01 Jack
02 Mack
03 Pham
…
```

Console output:

```
Student ID: 03, Name: Pham.
```

Writing to a File

Files - Writing to a File

- Use *write(str)* function, to write strings to file.
  - The file must be opened for writing (‘w’) or appending (‘a’).
  - ‘w’ truncates the file first
  - ‘a’ not truncating, places file pointer at the end of the file.

```
outputFile = open(filePath, "w")
outputFile.write("Hello World!\n")
outputFile.close()
```

Opening file for writing, can use ‘a’ for append

When writing to the file, you must add in the new lines

Closing the file

Closing a File

Files - Closing a File

- The previous example uses *close()* to close a file. Why close a file?
- Once a file is opened by a program, Python maintains connection to the file that generally prevents other programs from modifying, deleting, or moving the file.
- **If you are reading from a file** you don't want it to change because of some other program.
- When your program exit the file handle will be released automatically; but it is good practice to explicitly release file once you are done reading, while your program runs to let others get access to it.

Files - Closing a File

- **If you are writing to a file:** it is important to know that writing does not occur instantly. Python maintains a buffer where it stores the strings that are ought to be written to the file. Python and the OS coordinate the best time to **flush** this data and push it into the actual file area in memory.
- If your program crashes or terminates without closing a file, the **flush** action may not have occurred; Python 3 will close and flush buffers when terminating normally, but a programmer should not depend on this.

Files - Closing a File

- Also, opening files, reading from them and writing to them consume memory. Having unnecessarily-opened files means some memory is being used when it can be freed.

- CLOSE YOUR FILES!!!

Standard Input/Output/Error

Files - Standard Input, Output and Error

- We have been using files since the first program that we wrote.
- When you run a Python program, the interpreter opens up three streams, standard output, input, and error. When a program ends, it closes these streams.
- These streams are used for display output, and accepting inputs from/to Python programs.

Files - Standard Input, Output and Error

- **stdout**: is the file handler for the **St**andar**d Out**put file.
  - It is opened and closed automatically when the program starts and ends.
  - Any values written to it are displayed on the screen.
  - The file handler is accessible via the sys package: **sys.stdout**
  - You can write to *stdout* using the write method, just like any other file.
    - You have been writing to it using the print function:

```
print("Hello World!")
```

is the same as

```
sys.stdout.write("Hello World!\n")
```

Files - Standard Input, Output and Error

- **stdin**: is the file handler for the **St**andar**d In**put file.
  - It is opened and closed automatically when the program starts and ends.
  - Values can be written to it when input() or similar function is called. Python's interpreter passes these values to the caller.
  - The file handler is accessible via the sys package: **sys.stdin**
  - The *input()* function is equivalent to calling:

```
sys.stdin.readline().rstrip()
```

Files - Standard Input, Output and Error

- **stderr**: is the file handler for the **St**andar**d Err**or file.
  - It is opened and closed automatically when the program starts and ends.
  - Any values written to it are displayed on the screen.
  - Intended for displaying error messages instead of program output
  - Allows us to redirect program output separately from error messages
  - Useful for debugging
  - The file handler is accessible via the sys package: **sys.stderr**
  - You can write to *stderr* using the write method, just like any other file.
    - You can also use the print function:

```
print("ERROR!", file=sys.stderr)
```

is the same as

```
sys.stderr.write("ERROR!\n")
```

Onward to … exceptions.

Jonathan Hudson
jwhudson@ucalgary.ca
https://cspages.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY