Module 10: File input and Output

Topics:

- File input and output

Readings: ThinkP 8, 12, 14

Screen output and keyboard input

Our programs get their data from

- function parameter values,
- state variables declared in our program, or
- data entered by the user at the keyboard

and have displayed results to the screen.

Programs reading information from or writing to a file would be quite useful.

Input/Output beyond the screen

- Computers store data in files
- Files are *persistent*: data exists after your program ends
- Files created by one program can be used by other programs
- We will see how our programs can
  - read input from files instead of from the keyboard
  - write results to files instead of to the screen

Creating a Text File for Reading

- In CS116, we are working with text files only.
- How to create a text file?
  - In an editor, save as a text file
  - Wing IDE, "save as" -> choose option for "plain text"
  - Not:
    - **.doc, .docx, .pdf, .rtf**
    - These are all binary formats.
  - Any editor can be used to read/edit a plain text file.

Pattern for using a file in Python

- Find the file
- Open the file
- Access the file
  - Write to the file, or
  - Read from the file
  - *Cannot read from a file being written to*
  - *Cannot write to a file being read from*
- Close the file

Step 1: Finding a file

- *Easiest Solution*: ensure that the file being accessed is in the same folder as the program using it (the *active* folder or directory)
- More general solution
  - **`os`** module contains functions for interacting with the computer's operating system
  - **`os.getcwd()`** → name of current directory
  - **`os.listdir(os.getcwd())`** → list of names of files in current directory
  - **`os.chdir(dir_name)`** → changes the current directory to the name given by **`dir_name`**

Step 2: Opening a file

- **`file`** module gives us access to files in the current directory
- **`file(filename, "r")`** or **`file(filename)`** opens the file named **`filename`** for reading
- **`file(filename, "w")`** creates the file named **`filename`** for writing.
  - Warning! If there is already a file named **`filename`**, its contents are erased before the new data is written. Be careful!

When opening files, things can go wrong …

- If the file cannot be found or cannot be opened in the desired mode, the program will have a run-time error
- Alternative:
  - "Guard" the file action by placing it inside a **`try-except`** block
  - If an error occurs in the **`try`** block, the code in the **`except`** block is executed right away
  - If no errors, then the code in the **`except`** block is not executed

Using **`try-except`** to avoid fatal file errors

```
## safe_open: str str -> (union file False)
## produces False if filename could not be
## opened, and produces open file object
## otherwise
def safe_open(filename, mode):
  try:
    f = file(filename, mode)
    return f
  except:
    print "File %s not opened" % filename
    return False
```

Be careful with `try-except`

- Any type of error in the `try` block will cause the `except` block code to be executed as soon as an error happens
- Be sure that the steps in the `except` block are suitable for all errors in the try block
- Suggestions:
  - Do not use `try-except` until you have debugged the code in the `try` block for other, avoidable errors
  - Do **not** use `try-except` as an alternative to an `if` statement

Step 3: Accessing files - reading

- `f.readline()`
  - Returns the next line from file `f`
  - Includes newline character
  - Returns the empty string when at end of file
- `f.readlines()`
  - Returns a list of strings containing each line from file `f`
  - Each string terminates with newline character (if present in file)
  - If file is very large, this may consume a lot of memory

Example: Processing a file of names

Suppose you have a file containing a collection of names, where each line contains a single name in the form

```
first_name (spaces) last_name
```

Write Python code to create a list of `Name` objects from the open file object called `names`.

A useful helper function

```
class Name:
    'fields: first, last'
    def __init__(self, first, last):
        self.first = first
        self.last = last

## str_name: str -> Name
## produces Name from s, where s has the
## form "first last"  or "first last\n"
def str_name(s):
    nameslist = s.split()
    return Name(nameslist[0], nameslist[1])
```

Example: Solution One

- Read and convert one name at a time

```
next_str = names.readline()
people = []
while (next_str != ""):
  next_name = str_name (next_str)
  people.append(next_name)
  next_str = names.readline()
```

Example: Solution Two

- Read all lines, then convert all strings

```
all_names = names.readlines()
people = map(str_name,all_names)
```

Step 3: Accessing files - writing

- **`f.write(s)`**
  - Appends the string s to the end of file `f`
  - Writes the newline character only if `s` includes it
- **`f.writelines(los)`**
  - Appends all the strings in `los` to the end of file `f`
  - Writes newline characters only for those strings in `los` which include it

Recall: If you open an existing file for writing, you lose the previous contents of that file.

Example: Write Names in the form **`last_name, first_name`**

```
out_file = file("reversed.txt","w")
for p in people:
   out_file.write("%s, %s\n" %
      (p.last, p.first))
```

Step 4: Closing files

- **`f.close()`**
  - Closes the file `f`
  - If you forget to close a file after writing, you may lose some data
  - You can no longer access a file after it has been closed

Template for reading from a file

```
input_file = file(filename, ‘r’)
## read file using
##    input_file.readline()
##        in a loop, or
##    input_file.readlines()
## Note: resulting strings
##       contain newline
input_file.close()
```

Template for writing to a file

```
output_file = file(filename, ‘w’)
## write to file using
##    output_file.write(s)
##        in a loop, or
##    output_file.writelines(los)
## Note: newlines are written only
##       if strings include them
output_file.close()
```

The Design Recipe and Files: Modifications

- Effects:
  - Both reading from and writing to a file should be included in the Effects statement
- Testing
  - Use **`check`** package

Testing File Input

```
# process_file: str -> (listof int)
def process_file(filename):
       f = file(filename, “r”)
       …
```

- Set up a test file of data

```
check.expect(
          'Q1T1',
          process_file("q1t1file.txt"),
          [2,4,6])
```

Testing File Output

- Create text files that look like the expected output but with *different* file names than the files your function creates.

```
check.set_file(actual, expected)
```

`actual` – name of file created by program

`expected` – name of file you created with the expected output

Testing File Output – Alternate form

```
check.set_file_exact(actual,
  expected)
```

`actual` – name of file created by program

`expected` – name of file you created with the expected output

More on Testing File Output

- Use the appropriate **`check`** function to test the produced value.
- This will compare the value produced by the function, as before.
- It will also compare file contents as indicated by the **`check.set_file or check.set_file_exact`** call.
  - **`set_file`** ignores whitespace when comparing file contents
  - **`set_file_exact`** considers whitespace

Testing with files: an example

```
# file_filter: str int[>=0, <=100] -> None
# Purpose: consumes string fname, representing a
#   filename, and an integer, minimum, between 0
#   and 100. Produces None.
# Effects: Reads integers (one per line) from the
#   file with name fname, and writes each of those
#   integers which is greater than minimum to a new
#   file, summary.txt
# Examples:
# If ex1.txt is empty, then
#   file_filter("ex1.txt", 1) will create an empty
#   file named summary.txt.
# If ex2.txt contains 35, 75, 50, 90 (one per line)
#   then file_filter("ex1.txt", 50) will create a
#   file named summary.txt containing 75, 90
#   (one per line)
```

```
def file_filter(fname, minimum):
    # Assume fname exists
    infile = file(fname, "r")
    lst = infile.readlines()
    infile.close()
    outfile = file("summary.txt", "w")
    for line in lst:
        if int(line.strip()) > minimum:
            outfile.write(line)
    outfile.close()
```

Sample Test Cases

```
# Test 1: empty file
# q3t1_input.txt contains nothing
check.set_file("summary.txt",
  "q3t1_expected.txt")
check.expect("Q3T1",
  file_filter("q3t1_input.txt", 40), None)

# Test 2: general case
# q3t2_input.txt contains thirty integers,
# equally split above and below 65
check.set_file("summary.txt",
  "q3t2_expected.txt")
check.expect("Q3T2",
  file_filter("q3t2_input.txt", 65), None)
```

What is a "file"?

- We have used the term “file” in multiple contexts:
  - A data file in the current directory containing data (text or numbers) for our program
  - A variable in our program corresponding to that data file
  - A Python module containing methods to access that file, using the variable in our program
- In reality, some physical device is used to store the letters or numbers in our data file

Storing data in a file

- Stored digitally
- Must be consistent across platforms
- Must be concise and easily manipulated
- Atomic data have standard forms
  - Integers
  - Floating point numbers
  - Characters

Storing Characters

- All letters in the Latin alphabet, numbers and symbols are given a standard code between 0 and 255 (called ASCII code)
  - Each code can be stored using 8 binary digits (called a byte)
  - A,B,C, …, Z are in consecutive locations
  - a,b,c,…, z are in consecutive locations
  - 0,1,2,…,9 are in consecutive locations
- Strings are stored in memory using the ASCII code for each character, in order

Helpful Python functions

- **`ord(c)`**
  - **`len(c) = 1`**
  - Produces the ASCII code for character **`c`**
  - e.g. **`ord('a') => 97, ord('\n') => 10`**
- **`chr(code)`**
  - **`0 <= code <= 255`**
  - Produces the string containing the character with the given `code`
  - e.g. **`chr(100) => 'd', chr(32) => ' '`**

Standards and Codes

- ASCII is not sufficient for representing all languages
- Larger codes are needed
  - Unicode is built into Python
  - Each character in Unicode requires 16 bits (2 bytes)
  - Other standards exist as well

Goals of Module 10

- Understand the process of reading from files
- Understand the process of writing to files
- Familiar with the concept of how strings are stored