

Module 10: File input and Output

Topics:

- Input from files
- Output to files

Readings: ThinkP 8, 12, 14

How functions can use data

Our functions get their data from:

- function parameter values,
- global constants declared outside our function, or
- data entered by the user at the keyboard

Our functions also:

- return calculated values,
- mutate parameters, or
- print information on 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 (you can use any editor that allows this).
 - Wing IDE, "save as" -> choose option for "plain text" or specify "**.txt**" suffix.
 - Not:
 - **.doc, .docx, .pdf, .rtf** endings
 - These are not the correct format.

Pattern for using a file in Python

- Find the file
- Open the file (using **open**)
- 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 using **close**

Step 1: Finding a file

- Ensure that the file being accessed is in the same folder as the program using it (the *active* folder or directory)
- The **os** module provides other ways of finding files, but we will always assume the file is in the same folder as the program using it.

Step 2: Opening a file

- `open(filename, "r")` or `open(filename)` opens the file named **filename** for reading
- `open(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!

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, fn, ln):
        self.first = fn
        self.last = ln

def str_name(s):
    '''returns Name from s, where s has the form
        "first last" or "first last\n".
        s may include extra whitespace
        str_name: Str -> Name'''
    nameslist = s.split()
    return Name(nameslist[0], nameslist[1])
```

Example: Solution One

- Read and convert one name at a time

```
names=open('names-small.txt', 'r')
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
- Continuing example ...

```
all_names = names.readlines()  
people =  
    list(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

```
# Continuing example ...
out = open("reversed.txt", "w")
for p in people:
    s = "{0.last}, {0.first} \n"
    out.write(s.format(p))
```

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
 - Required when writing a file
 - Good style when reading a file

Template for reading from a file

```
input_file = open(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 = open(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

- Purpose: Should include details about what is read from a file or written to a file
- Effects: Should mention reading from a file or writing to a file (don't need details here)
- Examples
- Testing
 - Use **check** package

Testing File Input

```
def process_file(filename):  
    '''process_file: Str->(listof Int)'''  
    f = open(filename, "r")  
    ...
```

- Set up a test file of data, and include a comment describing contents of file

```
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_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 returned value.
- This will compare the value returned by the function, as before.
- It will also compare file contents as indicated by the **check.set_file_exact** call.

Testing with files: an example

```
def file_filter(fname, minimum):  
    '''opens the file called fname, reads in each  
        integer, and writes each integer>minimum  
        to a new file, "summary.txt".  
    Effects: Reads file called fname  
             Writes to a file called "summary.txt"  
    file_filter: String Int -> None  
    requires: 0 <= minimum <= 100  
    Examples:  
    If "empty.txt" is empty, then  
        file_filter("empty.txt", 1) will create an  
        empty file named summary.txt  
    If "eg2.txt" contains 35, 75, 50, 90 (one per  
        line) then file_filter("eg2.txt", 50) will  
        create a file named "summary.txt"  
        containing 75, 90 (one per line)'''
```

```
def file_filter(fname, minimum):  
    # Assume fname exists  
    infile = open(fname, "r")  
    lst = infile.readlines()  
    infile.close()  
    outfile = open("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_exact("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_exact("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
- 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`
 - Returns the Unicode code for character `c`
 - e.g. `ord('a') => 97`, `ord('\n') => 10`
- **`chr(code)`**
 - `0 <= code <= 255`
 - Returns 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
 - Most characters in Unicode require up to 32 bits (4 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