Introduction to:
Computers & Programming:
Input and Output (IO)

Adam Meyers

New York University

Summary

- What is Input and Ouput?
- What kinds of Input and Output have we covered so far?
  - *print* (to the console)
  - *input* (from the keyboard)
- File handling
  - input from files
  - output to files
  - Text files vs. 'pickled' binary files
- URL handling

Input

- Input is any information provided to the program
  - Keyboard input
  - Mouse input
  - File input
  - Sensor input (microphone, camera, photo cell, etc.)
- Output is any information (or effect) that a program produces:
  - sounds, lights, pictures, text, motion, etc.
  - on a screen, in a file, on a disk or tape, etc.

Types of Input Covered in This Class

- So Far
  - Input: keyboard input only
  - Output: graphical and text output transmitted to the computer screen
- This Unit expands our repertoire to include:
  - File Input – Python can read in the contents of files
  - File Output – Python can write text to files

Files

- File = Named Data Collection stored on memory device
  - Different types of data: text, binary, etc
  - Accessible by name or address
  - Has start and end point
  - Program can read, created, modified, (and do other things to) files
- Text file can be treated like a (big) string
  - Human readable
  - ASCII/UTF-8/etc. encoding
  - Can be plain text or can contain markup (e.g., html)
- Binary files: not human readable, usually require specific programs to read

Folders/Directories and Paths

- A Folder or a Directory is a named stored item that contains other folders and/or Files
- The root directory of a storage device:
  - no other directory contains it
  - it contains all other directories/files on that storage device.
- A sequence from directory to directory to directory … ending in a directory or file is called a path.
  - Each item *n* in the path (except the root) is contained by the *n-1* item.
  - There is at least one path from the root to every file & directory, i.e., paths can be used to identify/locate files
  - Each path uniquely identifies a single directory or file (ignoring short cuts, aka, symbolic links)

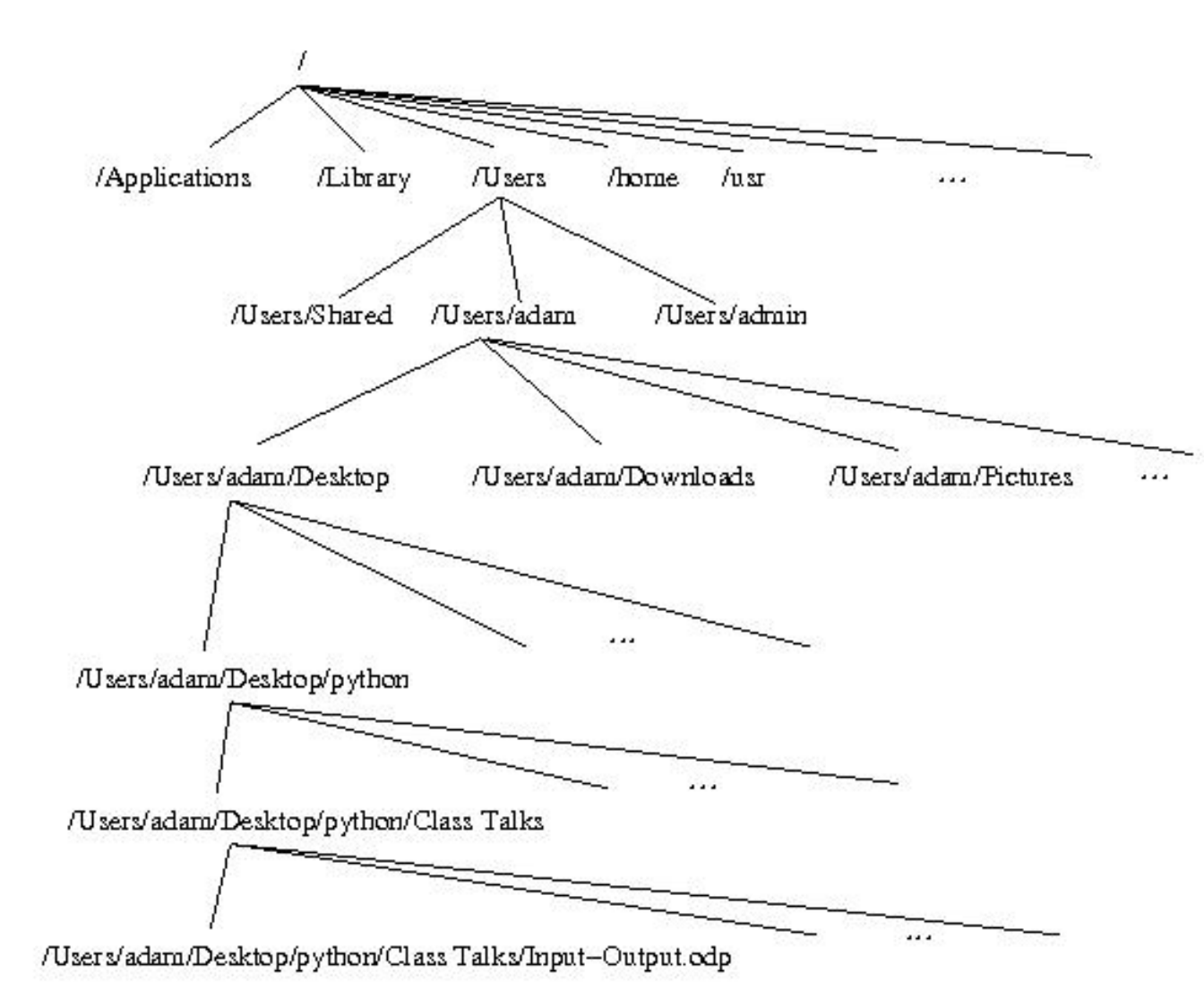
Slash Notation for Representing Paths

- Unix operating systems (linux, Apple, Android, etc.) use the forward slash to connect directories in a path, e.g., the path for this file could be:

  /Users/Adam/Desktop/Class Talks/Input-Output.odp

- MSDOS, Windows and related systems use the backslash instead \
- The root directory in UNIX systems is labeled /
- In Windows it is a letter and a colon, e.g., C:

The Directory Tree Including this File



Path Conventions

- The full pathname of a file is the path from the root to that file
- A relative pathname of a file is a path from some other point to that file
- Commonly, paths are described relative to some working directory, commonly called the “current working directory” or the “present working directory”

The os module

- Interface between python and Operating System
- http://docs.python.org/py3k/library/os.html
- For performing OS-dependent operations
  - handling files, checking for system features, getting root or administrative permission, etc.
- As with other modules, needs to be imported
  - import os, help(os), etc.
- Other system info is in platform module
  - e.g., platform.system() distinguishes Windows, Apple (Darwin), linux, etc.
  - Most of this info would not be needed for the types of programs we can reasonably expect to write this term

Global Variables from the os Module

- os.name
  - 'posix' (linux, current apple) or 'nt' (most Windows)
  - Others: 'os2', 'ce' ,'java','mac'(old Apple, I think)
- os.environ – all environmental variables (as a dictionary)
- os.linesep – '\n' for most systems, '\r\n' for Windows
- os.sep – '/' for most systems, '\\' for Windows

Listing, Renaming & Removing paths and Creating Directories

- os.getcwd() – gets current working directory
- os.listdir(*directory*) – gets children of *directory*
- os.chdir(*path*) – change current working directory to *path*
- os.mkdir(*path*) – make directory called *path*
- os.remove(*path*) – remove file (not directory) called *path*
- os.rmdir(*path*) – remove directory (not file) called *path*
- os.rename(*oldpath*,*newpath*) – rename (or move) *oldpath* to *newpath*
- os.path.isfile(*path*), os.path.isdir(*path*) – Boolean functions indicating if a particular pathname refers to a real file/directory
- os.system(command) – execute a terminal command as indicated by the string *command*

File Permissions

- There are certain directories and files that require root or administrative permission to open or read.
- It is possible to view and change these permission properties.
- For simplicity, we are only going to deal with files which our user has permission to create, remove and/or change

Files and Streams

- A stream is a continuous block of data ending in EOF (end of file character)
  - A computer program can read a stream
  - A computer can write to a stream
  - Other similar operations (e.g., append) are also possible
- An input stream can be created (opened) containing data found in a file. A program can then read data from this stream.
- A program can create (open) an output stream and add (write) data to it. When the stream is closed, the data in the stream is written to the file. This writing can either overwrite an existing file or create a new one.
- Other streams exist including Input/Output in a command terminal
  - standard input (the words you type)
  - standard output (what you see when words appear on the screen);
  - Others.

Reading and Writing to Files in Python

- instream = open(path, 'r')
  - creates an input stream containing the contents of the file named *path* and makes it the value of the variable *instream*.
- outstream = open(path,'w')
  - creates an output stream for writing data and set the variable *outstream* to this stream. Path names the file that will be created when this stream is closed (a previous file with that name would be overwritten).
- When the program is finished with a stream, it should close it as follows:
  - stream.close()
  - If *stream* is an output stream, a file is created or overwritten

Options for open(stream)

- Direction
  - r – read (previous slide)
  - w – write (previous slide)
  - a – append (add to the end)
  - + – open for read and write
- File Type
  - b – binary mode
  - t – text mode (default)

Sample function that reads and prints a text file

- def read_story(file):

```
    story = open(file,'r')
    for line in story:
        print(line,end='')
    story.close()
```

  - IO-examples.py
  - read_story('/Users/adam/Documents/short_story.txt')
  - read_story('short_story.txt')
- The file 'short_story'.txt – is in the current working directory
  - os.getcwd() → '/Users/adam/Documents'
  - os.listdir(os.getcwd()) → a big list of files
  - 'short_story.txt' in os.listdir(os.getcwd()) → True

To Get this Working On Your Machine

- You need to put a plain text file in the current working directory and/or use an absolute path name
- Windows paths are different
    - My windows cwd is 'C:\\Python31'
    - There are backslash instead of slashes
    - By default the file system does not display the file type (.txt)
        - So we have to be extra careful that we have the right filename
- In Python string, backslashes are indicated by using 2 backslashes
    - For Windows, it may be convenient to use the notation for a 'raw' string:

      r'C:\Python31\A Short Story.txt'
- Thus, on my Windows machine, the following commands work:
    - read_story('A Short Story.txt')
    - read_story('C:\\Python31\\A Short Story.txt')
    - read_story(r'C:\Python3\A Short Story.txt')

More about reading file

- The *for* loop treats the input stream (story) as a sequence of lines, each line being a string.

```
for line in story:
    print(line,end='')
```

- The *print* function does not require a newline after each string
  - Each line is a string that ends with '/n' for all UNIX systems (Apple, Linux) and with '/r/n' in Windows
  - In fact, it would print an extra line without *end=''*
- At the beginning of the read_story function, we open a stream which we call *story* as follows:

```
story = open(filename, 'r'))
```

- At the end of the function we close the stream as follows

```
story.close()
```

Function that Writes User Input to a File

- def take_dictation(outfile):

```
    dictation = open(outfile,'w')
    line = 'Empty'
    while (not (line == '')):
        line = input('Please give next line or hit enter if you are done. ')
        dictation.write(line+os.linesep)
    dictation.close()
```

- take_dictation('class_notes-Tues-11-23.txt')
- In IO-examples.py

Notes about Dictation Function

- Since I did not provide an absolute path, I know that the file will be in the current working directory.
- os.getcwd() → identifies the cwd
- The output file ('class_notes-Tues-11-23.txt') is located there.
- The function initializes an output stream using the 'w' (write) option of *open*
- The variable *line* is initialized as 'Empty'
- Then a while loop keeps going as long as line is not equal to the empty string.
- This kind of while loop is called a sentinel loop because we use a sentinel string (the empty string) to indicate when it is done.

More on the Dictation Function

- Other sentinel strings are possible.
  - The empty string is not ideal as the user might press enter by accident.
  - Perhaps, an explicit **stop** would make sure the user only stops when they mean to do so.
- The while loop prints each user input on a newline using the function (method) called write which is specific to streams.
- An end of line sequence is added to the end of the string.
  - We use the global variable *os.linesep* so the same program can be run on any platform (Apple, Windows, Linux, …)
- After the loop, we close the stream (and it writes to the file)

A Simple Spam Filter

- There are a bunch of email messages stored as files in a directory
- One at a time, the program reads these files and checks to see which ones pass a spam test.
- If the test says a file is spam, the program moves it into the *spam* directory, otherwise the program moves it into the *to-read* directory.
- In IO-examples.py

The implemented version of filter_spam

- Function call: filter_spam('letters','spam','to-read')
  - Sorts through the files in 'letters' and distributes them to 'spam' and 'to-read'.
  - Function call assumes that all directories are subdirectories of cwd
- filter_spam uses objects from the *os* package
  - os.path.isdir() – checks to see if the output directories exist
  - os.path.mkdir() – makes the directories if they don't exist
  - os.rename(file,destination) – 2 equivalent interpretations
    - moves a file from one path to another path
    - renames a file from one pathname to another
  - os.sep –global variable – '/' for UNIX and '\\' for Windows
- The function *is_spam* determines which directory a file is moved to

The function: *is_spam*

- A function that returns True or False
- The current version returns True if:
  - The file is too big (more than 25K bytes)
    - Is uses the *.st_size* slot of the object *os.stat(file)*
  - Or The subject line has no lowercase letter
    - Uses regular expressions to:
      - Identify the subject line
      - Find subjects with no lowercase letters
      - Introduces [a-z] to represent any letter between a and z
  - Or the subject line has a word from a list of spam words

A State-of-the-Art Version of *is_spam* Might have the Following Features

- It might pay attention to more of the letter than just the subject line (the email address, the body of the letter)
- It might look for (characteristics of) images and weird character sets
- It would probably incorporate large statistics on words that are more likely to be found in spam than in normal emails (it would not use a simple list)
- It would combine statistics, rather than basing the determination on the presence/absence of items in a list

More about Spam Program

- For simplicity, we treated letters as files
  - Actually the important issue is that they are streams, a more general concept that includes both files, letters, transmissions of different kinds, etc.
- This program needed to use the *os* package
  - In order to be platform independent
  - The specifics of file handling largely depend on the computer, operating system, etc. that you are using
- Weird Characters were also a factor
  - open(file,'r', encoding='utf-8', errors='ignore')
    - Encoding ensures that most characters are accepted
    - Errors='ignore' makes it so the program does not bomb on bad characters (important for email which mixes text and binary)

Application for Class Discussion: Reading/Writing Phonebook

- In IO-examples.py, there is a simple version of our application using a dictionary as a phonebook
  - def add_to_phonebook (name,phonenumber):

```
my_phonebook[name] = phonenumber
```

  - def interactive_add_to_phonebook ():

```
name = input('Enter New Contact: ')
number = input('Enter New Phone Number: ')
add_to_phonebook(name,number)
```

- Given a file which lists names and phone numbers in 2 columns, How can we input the file and use it to update our phonebook?
- How can we overwrite the file to contain the new phonebook?

Binary Files

- For our purposes, a binary file is any non-text file:
  - exe, jpg, gif, mp3, etc.
- The *open* function can read them using mode 'br' and write to them using mode 'bw'
- Pickling is a Python process for saving python data in binary form and retrieving it
  - Import pickle ## loads the pickle module
  - pickle.dump(python_object,outstream)

    ## sends python_object to the output stream *outstream*
  - *Instream = pickle.load(pickled_file)*
    - ## creates a stream *instream* with the contents of *pickled_file*

Processing Webpages

- import urllib.request
- Loads module for creating streams by reading in webpages
- Note that these streams will have many of the same properties as file streams
  - E.g., they can be treated as lists of strings
- http://docs.python.org/py3k/library/urllib.request

An Alternative to Opening and Closing Streams

- with open(filename,'r') as instream:

```
    for line in instream:
        print(line,end='')
```

- This is equivalent to:

```
    instream = open(filename,'r')
    for line in instream:
        print(line,end='')
    instream.close()
```

- The advantage is that it makes sure you close the streams that you open

Homework 1

- Read Chapter 8
- Start with the phonebook program that we wrote in class and adapt it to work with files.
- The program should be able to load in a phonebook file and create a phonebook dictionary in python.
- It should be possible to add additional names and phone numbers.
- It should be possible to save the new phonebook as a file (that can be loaded in again).

Homework 2

- It should be possible to reverse the dictionary (using the function we wrote in class).
- It should be possible to save the reverse dictionary as a file.
- The program in class could handle cases when two or more people shared the same phone number. Make sure this works in your adapted version.
- Optional: Alter all programs so to handle cases when a single person has more than one phone number.

Grading Criteria

1. Does it work?
2. Does it do what is asked for?
3. Is the code easy to understand?
4. Is the code elegant?
5. Did you implement the extra feature, allowing for one person having multiple phone numbers?