CS 115 Lecture 18

Using files

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

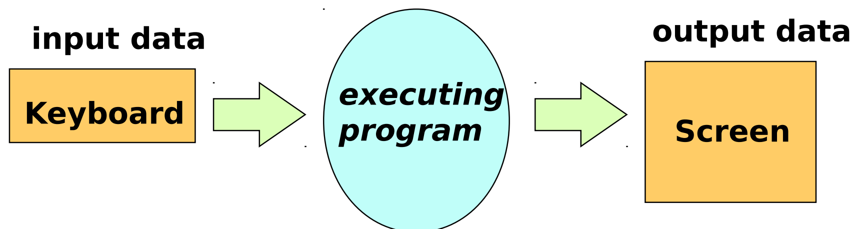
19 November 2015

Dealing with lots of data

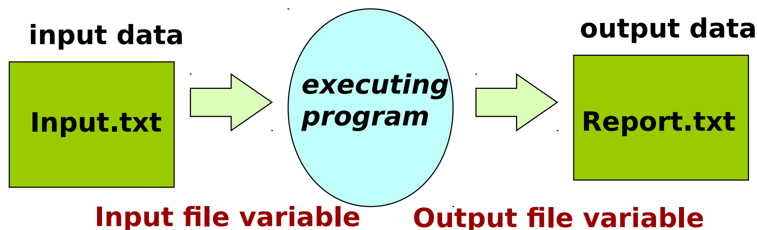
Some programs need a lot of data. What to do?

- Hard code it (write it in your source code)?
  That's hard for non-programmers to change.
- Ask the user to type it in each time?
  If it's a lot of data, your users will hate you.
- Do you have to type your source code every time you run it?
  No—you save it in a **file**.
- Why use files?
  - Easier to edit than source code.
    - Especially if you want to change it during a run.
  - Files persist across runs of your program.
    - And across reboots of your operating system.
    - Can save output for later use.
  - Can hold large amounts of data (more than fits in RAM).
  - Can use the same data as input to different programs.

Input/output with the user



I/O with files



Using files

As in other programs (word processors, IDEs, etc.), you must **open** a file before you can use it in your program.

- Create a **file object** in your program that represents the file on disk.
  - You can read from and/or write to the object.
  - Input-output from/to the file instead of the user.
- Syntax:

```
fileobj = open(filename, "r") # r for reading
fileobj = open(filename) # default is reading
```

  - `fileobj` is a variable that will hold the file object
  - `filename` is a string that names the file.
    - By default, looks for that file in the current directory.
    - You can specify an absolute path instead: `open("C:\\Users\\me\\input.txt")`
    - Don't do this in your 115 programs: your TA probably uses different directories.
- Can also open a file for writing:

```
fileobj = open(filename, "w") # w for write
```

IOError

If we are trying to read from a file, what can go wrong?

- Maybe the file isn't there.
  - Or it's there, but you don't have permissions to access it.
  - Or you do, but then your hard drive crashes.
  - Or it's on Dropbox/OneDrive and your network connection drops.
- In these situations, opening a file raises a `IOError` exception.
  - Renamed to `OSError` in Python 3.4.
- You can catch the exception just like any other.
  - But there's no point in trying again with the same filename.
  - Maybe ask the user for a new filename.

Re-prompting for a filename

```
ok = False
while not ok:
    try:
        fn = input("Enter a file name:  ")
        infile = open(fn, "r")
        ok = True
    except IOError:
        print("Could not open", fn)
```

Looping over the lines in a file

The simplest way to use an input file once you have opened it:

- Loop over the lines of the file.
- A file object can be used as a sequence of lines:
  `for line in file:`
  - Each line is a string.
  - `file` should be a file object, *not* a filename.
- Beware: the line ends in a newline character!
  - You might need to use `strip` or `rstrip`.
- When you're finished with the file, **close** it:
  `file.close()`
  - Frees up resources associated with the file.
  - If you don't, the file will take up memory until the program exits.
  - More on this later.
- `readfile-for.py`

Text files: characters and bytes

Files are stored on disk as a sequence of **bytes**.

- A byte is a collection of eight bits (ones or zeros)
  - Can represent a number from 0 to 255.
- In **text files**, bytes are used to encode characters.
- An **encoding** says how to translate between bytes and characters.
  - ASCII: one byte, one character—more than enough for English.
  - Latin-1, KOI8-R, . . . : Use the leftover numbers for more characters.
    - But 256 characters is not enough for CJK.
  - Unicode: more than 256 different characters.
    - So you need multiple bytes per character.
    - UTF-8, UCS-4, UTF-16: different encodings of Unicode.
    - UTF-8 is ASCII-compatible, so is the most commonly used.
    - (It's the default for text files in Python).
- **Text file**: stores a sequence of **characters**.
- **Binary file**: stores a sequence of **bytes**.
  - The difference is just a matter of perspective!

Text files: lines

So if a text file is just a sequence of characters, what is a line?

- A sequence of characters. . .
- There's one character that can't appear in the middle of a line.
  - The newline character!
  - Newline (`'\n'`) is the **line delimiter**.
    - (Technically it's a little more complex on Windows, but Python mostly hides that complexity.)
- What would two newlines in a row mean?
  - There's an empty line between them.
- So this file:

  ```
  Hello, world.

  How's it going?
  ```

  would look like:

  ```
  Hello, world.\n\nHow's it going?\n
  ```

Sequential and random access

- **Sequential access**: reading (or writing) the file in order starting from the beginning.
  - Like a for loop.
  - Read the first line first, then the second line, etc.
- **Random access**: reading or writing out of order.
  - "Go to byte 7563 and put a 1 there."
  - Like lists: we can say `mylist[5]` without having to go through indices 0 through 4 first.
- Random access doesn't work that well with text files.
  - Bytes don't always match up with characters.
  - And they definitely don't match up well with lines... At what byte number does line 10 start?
    - You'd have to go through the lines sequentially and count!
- **Text files**: usually **sequential access**.
- **Binary files**: either sequential or **random access**.

Reading a line at a time

- We already saw one way to read a line at a time:
  `for line in file:`
- This technique is very useful, but a little inflexible:
  - It only lets us use one line per iteration.
  - But what if our file looked like this?
    ```
    Student 1
    Score 1
    ...
    ```
- The `readline` method lets you control reading more precisely.
  `line = infile.readline()`
- This reads a **single line** from the file.
  - Returns a string, *including* the newline at the end.
  - The next time you input, you'll get the *next* line.
    - Even if you use a different input technique next time.
- `readfile-readline.py`

Buffers and the file pointer

- When you open a file, Python and the OS set up a **buffer** for the file.
  - A piece of memory that holds some of the file's data before you need it.
  - Why? Disks are much much slower than memory.
  - And disks are faster if they read large chunks.
- Where have you seen buffers before?
  - Youtube! "Video buffering..."
  - Same reason: memory is faster than the network.
  - Keyboard buffer: "type-ahead"
- When you call `readline` etc., that gets its data from the buffer.
  - If the program asks for data that isn't in the buffer yet,
    the OS re-fills the buffer with new data from the file.

The buffer holds data our program has already read...
...and data it hasn't read yet.

- How does it tell the difference?
- A **file pointer** indicates the beginning of the unread part.
  - Starts out at the beginning of the file (except in append mode).
- When you do a sequential read, like `readline`:
  - It starts reading at the location of the file pointer.
  - When it sees the newline, it advances the file pointer to the next byte.
  - So every read will get a different line.
  - Sequential access means the file pointer doesn't skip anything, and never moves backwards.

Reading a whole file at once

Python also gives us two methods that read in the whole file at once.

- That's much easier if we need to process the contents out of order.
  - Or if we need to process each line several times.
- But doesn't work if the file is bigger than memory.
- `readlines` gives us the whole file as a list of lines:

```
line_list = infile.readlines()
infile.close()
for line in line_list:
```

  - The lines still contain the newlines.
  - Why close it immediately? We already read everything!
  - After `readlines`, there's nothing left to read.
    - (at least with sequential input)
  - Technically, it gives you the *rest* of the file, after the file pointer.
    - Maybe you read the first line with `readline`, then called `readlines`.
- `readfile-readlines.py`

Reading a whole file

- `read` gives us the whole (rest of the) file as a single string.
  - Newlines and all.

    ```
    content = infile.read()
    infile.close()
    ```

  - As with `readlines`, you might as well close it immediately.
- What to do with the string?
  - You could use split:

    ```
    line_list = content.split("\n")
    ```

  - Unlike the others methods, this will give you a list *without newlines* (because `split` removes the delimiter)
- If the last line ended in a newline, there is an extra empty element:

  ```
  content = 'Hello\nWorld\n'
  content.split('\n')
  →[ 'Hello', 'World', '' ]
  ```

  - (Not just Python: some OSes and programs treat that as a blank line)
- `readfile-read.py`

Output files.

It's possible to write to files too.

- First, open a file for writing:
  `outfile = open("out.txt", "w") # w for write`
  - If the file doesn't exist, **creates it**.
  - If the file already exists, **truncates it**!!!
    - Cuts everything out the file to start over.
    - The old data is **lost forever**!
    - Opening for writing is both **creative** and **destructive**.
- You can also open a file for appending:
  `logfile = open("audit.log", "a") # a for append`
  - Adds to the end of an existing file—no truncation.
  - Will still create the file if it doesn't exist.
- Which to use? Do you want to add to the file or overwrite it?

Open summary

Mode	Letter	File exists	Doesn't exist
Read	r	OK	IOError
Write	w	Truncates the file	Creates the file
Append	a	OK (add to end)	Creates the file

Writing to an output file

There are two ways to write to an output file:

- You can use the same `print` function we've been using.
  - Just give an extra argument, `file = fileobj`:
    `print("Hello,", name, file = outfile)`
  - You can use all the functionality of print:
    - Printing strings, numbers, lists, etc.
    - `sep = ...`, `end = ...`
- You can also write a string with the `write` method:
  - Takes a *string* argument (nothing else!)
    `outfile.write("Hello, world!\n")`
  - Does not automatically add a newline!
  - Why would you ever use this instead of print?
    - That's how `print` is implemented!
  - `outfile-write.py`

Closing files.

- You should **close** a file when you are finished with it.
  `outfile.close()`
  - Frees resources (the file buffer!)
- Closing files is even more important for output files:
  - The buffer isn't written to disk until either it's full, or the file is closed.
    - Until you close the file, other programs can't see the new data.
    - And what if your program crashes?
  - Allows the file to be used safely by other programs.
    - Two programs reading is fine, but read + write can be a problem.
    - On Windows especially: "This file is in use by another application."
- `outfile.py`
- When to close a file?
  - As soon as know you won't have to read/write it again.
  - Immediately after your loop.
  - With `read` or `readlines`, immediately after reading.

Files versus lists

- Files are:
  - Permanent.
  - Bigger (must fit on disk).
  - Slower.
  - Sequential-access.
- Lists are:
  - Temporary.
  - Smaller (must fit in memory).
  - Faster.
  - Random-access.