Australian
National
University

COMP1730/COMP6730

Programming for Scientists

Input/Output and files

Outline

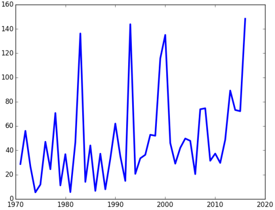
- Input and output
- The basics of **reading and writing** (text) **files**
- File system from a programmer’s perspective

I/O: Input and Output

- A (common) way for a program to interact with the outside world
  - **Input**: reading data (keyboard, files, network)
  - **Output**: writing data (screen, files, network)
- **Scientific programs** typically have to process and/or generate **large amounts of data**
- Today’s lecture will be mostly focused on **reading data from/writing data to files**, as this is the most common way to handle large volumes of data in scientific computing

```
2016,  07,  01,   2.0,  1,  Y
2016,  07,  02,   0.0,  1,  Y
2016,  07,  03,   0.0,  1,  Y
2016,  07,  04,   0.0,   ,  Y
2016,  07,  05,   4.4,  1,  Y
2016,  07,  06,  15.4,  1,  Y
2016,  07,  07,   1.0,  1,  Y
2016,  07,  08,   0.0,  1,  Y
2016,  07,  09,   4.2,  1,  Y
2016,  07,  10,   0.0,  1,  Y
2016,  07,  11,  10.4,  1,  Y
```

⇒



What is a file?

- A **file** is a collection of data stored on secondary storage (e.g., hard disk, USB pen, etc.)
- A program can **open** a file to read/write data from/to it
- Data in files is stored as a sequence of **bytes** (a byte as an integer b such that $0 \leq b \leq 255$)
- How this sequence of bytes has to be interpreted is defined by the so-called **format of the file**
- A program reading a file **must be aware** of the file format and follow the rules specified by the format in order to correctly interpret the data stored in the file
- Examples of file formats: text files, word processing (e.g. `docx`), image (e.g., `png`), music (e.g. `mp3`), and PDF files
- For simplicity, in this course, we restrict ourselves to **text files**

What is a text file?

- A sequence of printable characters (e.g., numbers, letters of the alphabet, spaces, punctuation signs, control characters, etc.)
- Characters are encoded using a character encoding format (roughly speaking, a mapping between characters and numbers)
- Examples of character encoding formats are: ASCII, UTF-8
- Fortunately, as programmers, we do not have to worry about character encoding formats, as Python takes care of this for us
- **IMPORTANT NOTE**: apart from the usual characters, text files typically also also contain **control characters**
- Examples of control characters: newline character (denoted symbolically as `\n`) or tab character (denoted symbolically as `\t`)
- Python programs are examples of text files

Reading text files

- Basic recipes for reading text files in Python are best illustrated through example
- We will work with a text file called `bom_monthly_mean_max_temp.tsv` (available at course web)
- The file contains **true temperature data** gathered by Bureau of Meteorology using a climate station located at Melbourne Olympic park
- File extension, i.e., `.tsv`, stands for **tab-separated values**
- This refers to the format of the text file, other examples of text file formats are `.csv` (**comma-separated values**) (Lecture 12!)

Text file format

- Before writing **any** program that reads a text file, **we must know the file format**, i.e., how the contents of the file are organized
- Structure of text in the file **greatly influences** the code that we need to write in order to appropriately read the file
- The first 5 lines of our example text file are as follows:

```
BoM station number Year Month Mean maximum temperature (C)
086338 2013 06 14.9
086338 2013 07 15.7
086338 2013 08 16.3
086338 2013 09 19.5
...
```

Text file format (cont.)

```
BoM station number Year Month Mean maximum temperature (C)
086338  2013 06 14.9
086338  2013 07 15.7
086338  2013 08 16.3
086338  2013 09 19.5
...
```

- The file presents a tabular structure with **4 columns** (although you cannot see it in slide, columns separated by tab characters)
- The first line is just a comment line with a human-readable description of the data in each column
- The actual data starts from the second line on
- The 4th column stores, in Celsius degrees, the monthly average of all daily maximum temperatures for the year and month combination given in the 2nd and 3rd columns

File objects

- To read a file, we first need to “**open**” the file
- This is achieved using the `open` function:

```
>> fin = open("bom_monthly_mean_max_temp.tsv","r")
```

- The first argument to `open` is a string with the **file name** (to be more precise, the “path” of the file, more on this later)
- The second argument specifies the so-called file **access mode**
- Access mode `"r"` denotes that we want to open the file for reading (read-only) mode
- The object returned by `open` is called a **file object**
- The file object is our interface to the file: all reading operations are done through methods of this object
- `fin` is a common name for a file object (short for “file input”)
- Once we finish processing the file, we have to close it using `fin.close()`

Reading operations

- Once we have opened the file, we can read its contents
- All reading operations are done through methods of the file object
- All reading operations return **strings**. Thus, we have to convert it to appropriate type (e.g., `int`, `float`) if needed
- Example: `readline()` method reads characters from the file until it gets to a newline and returns the result as a string

```
>>> fin = open("bom_monthly_mean_max_temp.tsv","r")
>>> first_line = fin.readline()
>>> first_line
’BoM station number\tYear\tMonth\tMean maximum temperature (C)\n’
>>> second_line = fin.readline()
>>> second_line
’086338\t2013\t06\t14.9\n’
>>> fin.close()
```

- Note the newline (`\n`) and tab control characters (`\t`)

String methods recap

- String is a very powerful data type in Python which offers many different methods (run `help(str)` on the Python shell)
- Two methods which are particularly useful when reading formatted text files (e.g., TSV, CSV) are:
  - `strip` removes leading/trailing whitespace (including newlines)
  - `split` splits a string into list of strings using specified separator

```
>>> fin = open("bom_monthly_mean_max_temp.tsv","r")
>>> first_line = fin.readline() # skip first line
>>> second_line = fin.readline()
>>> second_line
'086338\t2013\t06\t14.9\n'
>>> second_line_wo_newline = second_line.strip()
>>> second_line_wo_newline
'086338\t2013\t06\t14.9'
>>> second_line_wo_newline.split("\t")
['086338', '2013', '06', '14.9']
>>> second_line_wo_newline.split("1")
['086338\t20', '3\t06\t', '4.9\n']
>>> fin.close()
```

The concept of file position

- A text file is a sequence of bytes (representing characters)
- The file object keeps track of where in the file to read next
  - The next read operation starts from the current position
- When a file is opened for reading, the starting position is 0 (beginning of the file)
- File position is **NOT** the line number (typical misconception)
- `fin.tell()` returns current file position

More on reading operations

- `fin.read(size)` reads at most *`size`* characters and returns them as a string (if `size < 0`, reads to end of file)
- If file position already past the last character of the file, `readline` and `read` return an empty string (useful for writing `while` loops)
- `fin.readlines()` reads all remaining lines of text returning them as a list of strings

Example on reading operations

Suppose the text file "notes.txt" contains:

```
First line
Second line
Last line
```

```
>>> fin = open("notes.txt", "r")
>>> fin.read(4)
’Firs’
>>> fin.readline()
’t line\n’
>>> fin.readlines()
[’Second line\n’, ’last line\n’]
>>> fin.readline() == ""
True
>>> fin.close()
```

Iterating through a file

- Python’s text file objects are **iterable**
- They are **NOT** sequence data types though!
- Iterating yields one line at time

```
fin = open("notes.txt", "r")
line_num = 1
for line in fin:
    print(line_num, ’:’, line)
    line_num = line_num + 1
fin.close()
```

Programming exercise

Write a program to compute, out of the file with temperature data from BOM, the yearly temperature average of all 12 monthly averages for year 2019

```python
fin = open("bom_monthly_mean_max_temp.tsv", "r")
temperature_sum=0.0
for line in fin:
    columns=line.split()
    # columns[1]: year, columns[2]: month, columns[3]: temperature
    if columns[1]=="2019":
        temperature_sum += float(columns[3])
avg = temperature_sum/12.0
fin.close()
print("Yearly temperature average for year 2019 is: " + str(avg))
```

Writing data to text files (write-only mode)

- Writing data to a text file requires the file object to be opened in a write access mode
- One of such modes is write-only access mode (denoted by `"w"`)

```
fout = open("notes.txt", "w")
```

- `fout` is a common name for a file object open in write-only access mode (short for “file output”)
- Creates a new empty file with name `"notes.txt"`
- **CAUTION:** if the file already exists in the file system, it erases the old contents of the file **without generating an error nor warning** (thus, one may loose data if not used carefully!)
- There are other file access modes (not covered here, go to Python documentation for more details)

Writing data to text files (write operations)

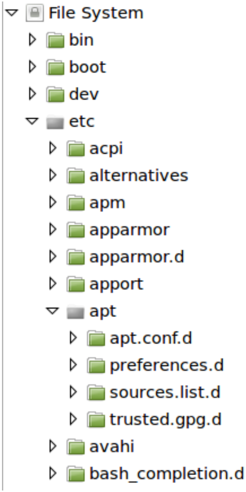
- Once we have opened a file in write-only access mode, we can start populating it with data
- `fout.write(string)` writes `string` to the file
- **IMPORTANT:** `fout.write(string)` does **NOT** add a newline to the end of `string`
- If one wants to write a newline at the end of `string`, one has to explicitly add it
- For example, we can use `fout.write(string+"\n")`

Writing text files (buffering)

- File objects typically have an I/O buffer in memory
  - Writing to the file object adds data to the buffer; when full, all data in it is written to the file (to “flush” the buffer into the file)
- Closing the file (i.e., `fout.close()`) flushes the buffer
  - If the program stops without closing an output file, the file may end up incomplete
- **Bottom-line**: always close the file when done!

File system (directory structure)

- Files on the computer are organised into **directories** (also known as **folders**)
- This is an abstraction provided by the Operating System (OS)
- The way this abstraction is presented to the programmer might differ among OSs (Windows versus macOS/Linux)
- The directory structure is typically **tree-like** (e.g., see figure on the right for an example of a directory structure in a Linux computer)



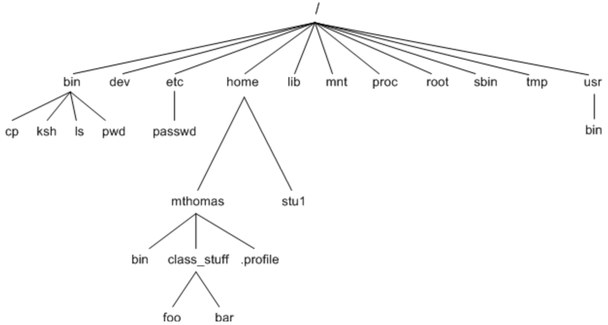
- File System
  - bin
  - boot
  - dev
  - etc
    - acpi
    - alternatives
    - apm
    - apparmor
    - apparmor.d
    - apport
    - apt
      - apt.conf.d
      - preferences.d
      - sources.list.d
      - trusted.gpg.d
    - avahi
    - bash_completion.d

The concept of path of a file

- A **path** is a string that identifies the location of a file in the directory structure
- When opening files from a program, we have to use paths to specify the location of the file we want to open
- The particular syntax of a path depends on the Operating System available on the computer (Windows, Linux/MacOS)

Example: Linux/MacOS

- In Linux/MacOS, a path is a string that contains a sequence of folder names separated each by the slash character `"/"`
- The last name in the path is the name of the file
- Observe that the path encodes the sequence of directories that one has to traverse in the tree in order to get to the file from the root of the tree
- Example: path of `.profile` file is `"/home/mthomas/.profile"`



Current working directory of a program

- Every running program has a current working directory (cwd)
- By default, the cwd is the directory from which the python interpreter was started (not necessarily the directory where the python program source file is located)
- The `os` Python module provides functions to get and modify the current working directory

```
>>> import os
>>> os.getcwd()
’/home/u1134396/git-repos/Teaching/COMP1730’
>> os.chdir("/home/u1134396/")
>> os.getcwd()
’/home/u1134396’
```

Absolute versus relative paths

- There are actually two kind of paths: **absolute** and **relative**
- A path is absolute if it starts with the character `"/"` (all examples so far) and relative otherwise
- A relative path is called relative because it depends (i.e., it is relative to) the current working directory
- Absolute paths do not depend on the current working directory
- **Note**: We can use `".."` in paths to denote the directory above (parent directory)

Examples of relative paths

```
/
:
├─home
:  :
:  ├─u123
:  :   ├─Desktop
:      ├─lab1
:      └─lab2
:
├─lib
:
```

Assume cwd is `"/home/u123/lab1"`

Example 1: `"prob1.py"` refers to `"/home/u123/lab1/prob1.py"`

Example 2: `"../lab2/prob1.py"` refers to `"/home/u123/lab2/prob1.py"`

Example 3: `"../../../lib/libbz2.so"` refers to `"/lib/libbz2.so"`

Programming exercise

Modify the code from Lecture 12 (COVID-19 vaccinations) such that data is read using programming instructions presented in this lecture instead of `csv` module

Take home messages

- Scientific programs need to process **large amounts of data**. These data are typically stored in files
- Many different kind of files, formats, etc. (focus here on text files)
- One needs to know the format of a text file (e.g., CSV, TSV) before writing the program that processes it (as the format greatly influences the programming instructions that you will write)
- Best practice: while coding, write `fin=open(...)` and `fin.close()` immediately before adding code in-between
- The file system presents a tree-like structure. We use paths (either absolute or relative) to specify the location of a file in the tree from the program