COMP1005 Week 5 Cheat Sheet

Lisa Luff

9/22/2020

Contents

Files

- Creates persistent data, that will remain after the program is complete
- Allows more versatility
- Accessed via their "path"
  - Default is within the current directory
  - If not within directory, include path with file name
    - '*/home/location1/location2/filename.filetype*'
- Some files will hold text, and others binary data
- When opening a file we can read, write or append to it
  - Defaults to 'r' (read)
- Need to create a file object to utilise the file contents
  - Use *filevariable* = open('*filename.filetype*', '*x*')
  - Where '*x*' is how to process the file/file modes

File modes

Read	Write	Append	Description
r	w	a	Read/write/append text files
rb	wb	ab	Read/write/append binary files
r+	w+	a+	Opens for reading and writing
rb+	wb+	ab+	Opens for reading and writing

- Need to close files safely
  - Use *filevariable*.close()
  - Use it immediately after assigning the file variable
  - This flushes any unwritten information

Reading Files

- Three types:
  - *filevariable*.read()
    - Reads the entire contents of the file as one block. The file is like one long array, and can be indexed as such.
  - *filevariable*.readline()
    - Reads one line at a time. A line is defined as the beginning of a string until \n. If you run this multiple times, it will move through each of the lines in the file
  - *filevariable.readlines()
    - Will read the entire contents of the file, separated into lines and stored as a list where each line is an element
    - The lines will include \n, so you can use .striplines() to remove these types of things

Writing to files

- Done with *filevariable*.write("*what you want to write*");
  filevariable.close()

Together

filename = open('*filename.filetype*', '*x*')
filevariable = *filename*.*y*()
filename.close()
* Where *y* is read(), readline(), readlines(), write(), etc.

CSV Files

- Reading CSV files:
  - CSV files contain rows separated by \n, and columns separated by commas
  - There is a csv package, but we will manually do it for now
  - How to do it manually:
    - Still open it the usual way to read
    - For a single line use readline()
    - Then separate the line with *splitvariable* = *filevariable*.split(‘,’), to create a list in order of everything between the commas
      - Use a loop if using readlines() to split each line
      - Use .strip() after .split(‘,’) to strip any thing like \n as you make each element, rather than using a secondary loop

- Writing CSV files:
  - Turn a list of items into a string separated by commas with:
    - *joinedvariable* = ‘,’.join(*listname*)
  - Then open it the usual way to write
  - *filevariable*.write(*joinedvariable*)

The Pythonic Way

- It is best practice to use **with** for file objects
- It will automatically close the file when you’re done, even if an exception occurs
- All together:
  with open(‘*filename*,’*x*’) **as** f:
  filevariable = f.*y*()

Binary Files

- Won’t cover this in this course
- Advantage:
  - Stored as binary, so much more compact
- Disadvantages:
  - We can’t read them directly
  - Unlikely to be able to fix them if they’re corrupted

Grids

- Breaks up a space into a multidimensional grid
- Each cell has one or more associated values
- The cells impact on each other over time
- General algorithm:

```
For each time_step
- For each row
-- For each column
--- Calculate the current value, and set up the next value
```

Neighbourhoods

- For a 2-D grid, the vonNeumann neighbourhood of a site is the set of cells directly North, South, East and West of the site and the site itself

NW	**N**	NE
W	***Site***	**E**
SW	**S**	SE

- The Moore neighbourhood adds NE, NW, SE and SW

NW	**N**	**NE**
W	***Site***	**E**
SW	**S**	**SE**

- The four or eight cells, not including the site, are the site's neighbours

Algorithm

- Each element(cell) of a grid is affected by the elements(cells) around it
  - Which cells affect it will depends on the neighbourhood type
- Need a for loop for how many times you are going to work through the grid
  - Then within that, a for loop to work through each row of the grid
    - ∗ And finally a for loop to work through each column of the grid
- Within the loops, you will have a formula to find the value of the individual cell selected by the for loops - this is the site
  - The formula will utilise a combination of the information from the neighbouring cells
    - ∗ Each neighbouring cell is specified in terms of the current position (site), as:

(ROW - 1, COL - 1)*formula*	(ROW - 1, COL)*formula*	(ROW - 1, COL + 1)*formula*
(ROW, COL - 1)*formula*	(ROW, COL)*formula* (***Site***)	(ROW, COL + 1)*formula*
(ROW + 1, COL - 1)*formula*	(ROW + 1, COL)*formula*	(ROW + 1, COL + 1)*formula*

- Example:

```
import matplotlib.pyplot as plt
import numpy as np

rows = $n$ columns = $m$ initialgrid = np.zeros($n$, $m$)

affectingpoint = value initialgrid[$n$, $m$] = affectingpoint

resultinggrid = np.zeroes($n$, $m$)

(Might need to exclude affecting point from below depending on situation) for timestep in range(x (number of time periods)):
- for n in range(0, $n$):
-- for m in range(0, $m$):
--- resultinggrid[$n$, $m$] = (initialgrid[n - 1, m - 1]function, etc. for all cells that are part of the neighbourhood)
- (If you need the affecting point at original value)
- resultinggrid[$n$, $m$] = affectingpoint
- initialgrid = resultinggrid

plt.imshow(resultinggrid, other)
plt.show
```

- **Be warned**: that the cells on the edges of the grid will not be affected the same as the rest
  - It might be useful to put a dead border around the grid and cut it from the final grid

List Comprehensions

- A pythonic approach that turns a multi-line for-loop into a one liner
- It is creating a list element by element itself, so you do not need to tell it to append the items to an empty list, or anything like that
- It works by looking at the elements as items
    - This means you do not need a range, as it will work through everything in the list
    - Also because we are working through items, not indexing them, the "i" or index reference is not a value between 0 and length-1, "i" itself is whatever that current item in the list is
        - For example:
          newlist = [3**i* for i in *list*]
          This is saying create a new list called *newlist*where each element in the list is the same element of *list* except with the value for that element in *list* multiplied by 3. As opposed to it being the index of the element being multiplied by 3.
- Basic syntax:

[*transformation* **iteration *filter***] OR
[*expression* **for item in list *if conditional***] (better phrasing)

- Equivalent to:

for item in list:
- if conditional:
-- *expression*

Unconditional List Comprehensions

- As a for loop:

```
list = [items]
newlist = [ ]

for item in list:
- newlist.append(do thing to item)
```

- As a list comprehension:

```
list = [items]
newlist = [do thing to item for item in list]
```

Nested Loops in List Comprehension

- You can do nested loops with list comprehension
- You start with the deepest nested loop within a set of brackets [], then the next level up will have the deeper nested loop and the outer loop within brackets [], etc.

- As a for loop:

```
listoflists = [listA[items], listB[items], etc.]
newlists = [ ]

for list in listoflists:
- for item in list:
-- newlist.append(do thing to item)
```

- As a list comprehension:

```
listoflists = [listA[items], listB[items], etc.]

newlist = [[do thing to item for item in list] for list in listoflists]
```

Conditional List Comprehensions

- As a for and if loop:

```
list = [items]
newlist = [ ]
```

```
for item in list:
- if condition:
-- newlist.append(do thing to item)
```

- As a list comprehension:

```
list = [items]
newlist = [do thing to item for item in list
-------------- if condition]
```

Multiple Condition List Comprehensions

- You can use if, elif and else in list comprehensions
- You will need to shift the conditions to the beginning of the list comprehension statement
- There is no direct elif, but you can create an elif statement using else and then if

- As a for and if, elif, else loop:
  list = [*items*]
  newlist = [ ]

```
for item in list:
- if condition:
-- newlist.append(do thing to item)
- elif condition:
-- newlist.append(do other thing to item)
- else:
-- do other thing
```

- As a list comprehension:

```
list = [items]
```

```
newlist = [do thing to item if condition
-------------- else do other thing to item if condition
-------------- else do other thing
-------------- for item in list]
```

Additional Notes

- Use interpolation='bilinear' to smooth out visualisation in graphs
  - Warning, this will no longer be the real data

- word[*element*] will look at element *element* in a string