COMP1005 Week 4 Cheat Sheet

Lisa Luff

8/25/2020

Contents

Multi-Dimensional Arrays

Defining 2-D Arrays

- Still need to use numpy

1. Build it from lists

- *variable* = np.array([[*x*, *y*], [*a*, *b*]])
- So you set up an array with a list of lists separated by commas
- Each list is one row, and the comma indicates the beginning of the next row

2. Build using array builders

- *variable* = np.zeros((*x*, *y*))
- This creates an array with x rows, and y columns
- Can use any array builder (ones, etc.)

3. Use meshgrid, which is similar to arange for 1-D arrays

- *x*,*y* = np.mgrid[0:5, 0:5]
- This creates two arrays
- x will have 5 rows, with row 0 containing 5 0s, row 1 containing 5 1s, etc.
- y will have 5 columns, with column 0 containing 5 0s, column 1 containing 5 1s, etc.
- All versions create a list within a list and will be shown as:
  - [[*x*, *y*], [*a*, *b*]]

Describing Arrays

Function	Use
print(x)	Will print as a list of lists
np.size(x)	Tells you how many elements in the array total
np.shape(x)	Tells you (a, b), where a is number of rows, and b is number of columns
len(x)	Tells you the length of the first dimension (number of rows)

Reshaping Arrays

- When reshaping, it will look at the elements as one long vector, regardless of current row and column structure, and break it up accordingly
- *array*.reshape(x, y) - Will break up *array* into x rows, and y columns

Indexing 2-D Arrays

- *array*[x, y] will call on the element where row x meets column y

Looping Through Arrays

- You need a loop per dimension
  - So you will need a loop that calls on either the row or column position
  - Then an internal loop to call on the alternate dimensions position
- The external loop will call on one vector of that dimension, then the internal loop will run through every element of that dimensions vector, and once complete, you will come back to the external loop and move onto the next vector in that dimension, and repeat.
- This can be done with
  for i in *array*[:,0]:
  for j in *array*[0,:]:
  do thing
  - This would run through every row (:), and ignore columns (0)
  - Then would ignore rows (0), and run through every column (:)
  - You can add specifiers to either side, but make sure you're only specifying for one dimension at a time

Looping the Pythonic Way

- Looping in python looks at each element of an array or list as an "item"
  - This means that you do not need a range, as it will look at each item in the array in turn
  - It also means that whatever you do to "item" will be to the item itself, not the value of the index, as we are technically not indexing

- You can use indexing by using enumerate()
  - And enumerate specifies that we are indexing the array
  - Doing it this way means we use the item system but can also reference the index values
    - Eg. for print(*newitem*, *indexvalue*)
  - It also means you can set the index value to start somewhere other than 0, while still indexing the entire array
    - Using enumerate(*array*, start = *number*)

- This can be simplified to:

1. Non-Indexing
   for *row* in *array*:
   - for *item* in *row*:
   -- *do thing to item*

2. Indexing
   for *rindex*, *row* in enumerate(*array*):
   - for *cindex*, *item* in enumerate(*row*):
   -- *do thing to item*

- In this case, rindex, and cindex represents the value of the index for the rows and columns respectively

Slicing 2-D Arrays

- It is the same method, you just put in a specification per dimension separated by commas and the output will by the intersection of the specifications
  - eg. *array*[1:, ::2], will give you where rows 1 to end, intersect with every second column

Working with Multi-Dimensional Arrays

- Operations and comparisons will still align element to equal element, so *array1*[*x*, *y*] will be aligned with *array2*[*x*, *y*]
  - This means that they need to be the same shape
- Element-wise functions will still act on every element of the array
- Array-wise functions can be used across the entire array, or you can slice it to be used for only one dimension of the array
  - eg. *array.function*() - Will combine the elements from the entire array
  - eg. *array*[:, 0].*function*() - Will only combine the elements from the rows in column 0

Matrices

- Again need to use numpy
- Set up using matrix() function
  1. Using lists, *variable* = matrix([[*x*, *y*], [*a*, *b*]])
  2. Using formatting like matlab, *variable* = matrix('*x y*; *a b*')
- Can then use matrix manipulations
  - Operations (+, *, **) will be matrix appropriate
  - Can use np.linalg (linear algebra package)
    - np.linalg.det(*array*) - Gives the determinant
    - *array*.T - Transposes a matrix
    - *x*, *y* = np.linalg.eig(*array*) - Will assign *x* the eigen value and *y* the eigen vector

Scipy

- A package that uses numpy arrays for:
  - Interpolation
  - Integraion
  - Optimisation
  - Image processing
  - Statistics
- Includes task-specific sub-modules

Scipy N-Dimensional Images

- scipy.ndimage
- We are using a sub-module, so you need to tell it to import this from within the package
  - from scipy import ndimage
- We plot the images using a similar function to matplotlib
  - plt.imshow(*imagename*)
  - Use a colourmap with
    - plt.imshow(*imagename*, cmap=plt.cm.*colourscale*)
- Plots the same as a normal plot, except that 0 for the y axis starts in the top left, rather than the bottom left
- Functions:
  - ndimage.shift(*imagename*, (*x*. *y*)) - Will have the top left corner at coordinates (x, y)
  - ndimage.rotate(*imagename*, x) - Will rotate the image n degrees anti-clockwise
  - *imagename*[*slice*, *slice*] - will crop the photo by keeping only what is within the specified slices for each dimention
  - To pixelate the image, just slice it with steps. The more steps, the more pixelated

Functions

- Set a function with def *function name*(*arguments*):
  - Then indent everything within the function by 4 spaces
  - There doesn't need to be any arguments, but you still need ()
  - return(*variable*) will be the output of the function

Methods

- Methods are a special type of function associated with a class/object
  - Relatively interchangable with functions
- Objects are datatypes that have associated data and operations
  - In Python everything is an object, which makes the two even more interchangable

Flow of Execution

1. Import statements
2. Function definitions
3. Set up variables
4. Input data
5. Process data
6. Output data

Writing Programs

- What to look for when creating your pseudo-code:
  - Decisions - if/elif/else
  - Iteration - while and for loops
  - Repeated code/tasks - functions
  - Data to store - variables (what kind of data)
  - Data to read in/print out - input and print calls

Python Modules and Packages

- You can create a module for functions you want to use in many programs

1. First you create a module *modulename*.py
2. Next you create some methods/functions within this; *functionname*()

- To use the functions, import *modulename* at the start of the program
  - Then use it with *modulename.funtionname*()
- You can group modules together into packages
  - A package is indicated with ___*packagename*___.py (two underscores either side)
  - It is a directory
  - You would use import the modules in a function with, import *packagename.modulename* as *pm*

Paths

- Rather than having all modules within the local directory'', the operating system uses a "PATH" variable which tells is a list of directories to look through when searching for something
  - Part of installtion is updating the path to include the new program
  - You can view the path in terminal with; echo $PATH
- If you write a program with a number of data manipulation functions in it, that you might want to use in other programs, and one main function that is what runs when the program is called on directly in terminal. You can make sure that main program doesn't automatically run when importing that program when utilising those data manipulation functions on their own in other programs by adding this to the original program:
  if ___name___ == '___main___':
  nameofmainfunction()

Plotting Multi-Dimensional Data

- Warning for colour maps: With spectrums, some colours are brighter than others, which can make the plot misleading
- Contour plots
  - plt.contourf(x, y, f(x, y), n, alpha=a, cmap=b)
    - Contour function is a 3 dimensional plot which is filled with colour based on b
    - x is x-axis value
    - y is y-axis value
    - f(x, y) is the function for the third dimension
    - With n + 1 levels/layers/depths of colour contrast
    - And a transparency (0.0 - 1.0)
  - plt.contour(x, y, f(x, y), n, colors='a, linewidth=b)
    - Contour is a 3 dimensional plot which has lines whose colour is based on a
    - The lines with have a width of b
- Scatter plots
  - plt.scatter(x, y, s=a, c=b, alpha=c)
    - Where a is the type of point to use
    - And b is the colour-scheme to use