ENGG1003 - Monday Week 7

Creating Your Own Libraries: Python Modules
2D Arrays

Sarah Johnson

University of Newcastle

19 April, 2021

Lecture Overview

1. Modules
2. 2D Arrays

The Story So Far

- variables and data types
- 1D arrays (using `numpy`)
- plotting (using `matplotlib`)
- flow control
- functions
- interpolation (using `scipy`)

Today we are going to extend two of these topics: by putting functions into modules, and by considering arrays with more than one dimension.

Modules

- A *module* is a python script saved as .py file
- A *package* is a collection of modules.
- Packages / modules shared with others are often called *libraries*.
- We have seen some very helpful libraries so far in ENGG1003. E.g. numpy, matplotlib.

Modules

- Any of the Python scripts we have written so far could be regarded as a module (though perhaps not a very useful one).
- e.g. ball.py from week 2 (LL Chapter 1):

```
1 v0 = 5
2 g = 9.81
3 t = 0.6
4 y = v0*t - 0.5*g*t**2
5 print(y)
```

Modules

- We can import the code ball.py

```
1 import ball
```

- The import statement would just cause this code to be executed and print out the value of y

Modules

- A more useful model might be ball_function.py from week 5:

```
1  def ball_height(v0, t):
2      g = 9.81
3      y = v0*t - 0.5*g*t**2
4      return y
5
6  v0 = 5
7  time1 = 0.6
8  height1 = ball_height(v0,time1)
9  time2 = 0.9
10 height2 = ball_height(v0,time2)
11 print('height 1: {}, height 2: {}'.format(height1,
       height2))
```

Modules

- We can import ball_function and use its function

```
1 import ball_function as bf
2
3 v0 = 5
4 time = 0.7
5 height = bf.ball_height(v0, time)
6 print('At time {} the height is {:5.3f}'.format(
      time,height))
```

- At the import statement, the code in ball_function.py is executed - printing out the values of height1 and height2 and loading in ball_height ready for use.

Modules

- Even better let's define a module ball_motion.py containing just functions

```
1 def ball_height(v0, t, g=9.81):
2     y = v0*t - 0.5*g*t**2
3     return y
4
5 def time_of_flight(v0, g=9.81):
6     y = 2*v0/g
7     return y
```

Modules

- We can import ball_motion

```
1 import ball_motion as ball
```

- When the code is executed the functions would be imported - but nothing would be printed out. However, the functions ball.ball_height and ball.time_of_flight would be available for use as needed in our code.

Modules

- We should add some documentation:

```
"""
Module for computing vertical motion of a ball
"""
def ball_height(v0, t, g=9.81):
    """
    Vertical position at time t
    """
    y = v0*t - 0.5*g*t**2
    return y

def time_of_flight(v0, g=9.81):
    """
    Time of flight of the ball
    """
    y = 2*v0/g
    return y
```

Modules

- Now we can import import ball_motion and get some information about it

```
1 import ball_motion as ball
2 help(ball)
```

- Let's run these live ...

2D Arrays

- A *scalar* is a single item (one element in memory)

$$s = 3.0$$

- A *1D array* or *vector* is a series of elements in memory

$$a = \begin{bmatrix} 3.0 & 1.9 & 4.8 & 6.2 \end{bmatrix}$$

- A *2D array* or *matrix* is rectangular array of items (still a series of elements in memory but indexed differently)

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- A 2D array is made up of horizontal 1D arrays called *rows*, or vertical 1-D arrays are called *columns*

2D Arrays

- We have seen many examples already of where 1D arrays are useful
- 2D arrays are useful for
  - storing/indexing objects which naturally have a grid structure, such as the the pixels in a picture
  - organising data sets e.g. storing the entire dataset from a csv file
  - working with matrices in linear algebra (which you will become very familiar with if you do MATH2320)
  - .....

Arrays in ENGG1003 use Numpy

- A *scalar* is a single integer, float, char, etc
- A *1D array* is a series of integers, floats, chars, etc
- A *2D array* is a series of integers, floats, chars, etc organised in a rectangular structure

* Note that Numpy arrays must contain all entries of the same type (e.g all integers or all floats etc.) This is not the case for other types of 1D or 2D objects in Python such as lists.

* Numpy can handle arrays of dimension up to 32D (but we won't be needing that functionality in ENGG1003)

Reminder: Creating Scalars and 1D Arrays

- Creating a scalar. Some examples:

```
s = 3.0
s = 2
s = np.pi
s = len(a)
```

- Creating a 1D array. Some examples:

```
a = np.zeros(4)                    # a = [0, 0, 0, 0]
a = np.linspace(0,3,4)             # a = [0, 1, 2, 3]
a = np.array([3.0,-1.9,4.8,6.2])
                                   # a = [3.0, -1.9, 4.8, 6.2]
```

Creating 2D Arrays

- Creating a 2D array. Some examples

```
1 m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
1 m = np.zeros([2,2])
```

$$m = \begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix}$$

Indexing Arrays

- Index a 1D array using $[i]$ where i is the index we want eg:

```
a = np.linspace(0,3,4)   # a = [0.0, 1.0, 2.0, 3.0]
s = a[2]                 # s = 2.0
a[0] = 4                 # a = [4.0, 1.0, 2.0, 3.0]
```

- Index a 2D array using $[r, c]$ where r is the row index and c is the column index eg:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
s = m[0,2]          # s = 3
s = m[2,1]          # s = 8
m[1,1] = 11         #[[1, 2, 3], [4, 11, 6], [7, 8, 9]]
```

Indexing Arrays

- We can also select whole rows or column using ':' which means 'all row/column entries'

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
    a = m[0, :]      # a = [1, 2, 3]
    a = m[:, 1]      # a = [2, 5, 8]
```

- Note: if you write m[1] Python will assume you mean m[1,:] and will return row 1

Indexing Arrays

- We can also select a subset of rows and/or columns of a 2D array to create a new array

```
1 m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2 a  = m[0, [0, 2] ]     #row 0 and cols 0 & 2
3 m2 = m[:, [0, 2] ]     #all rows and cols 0 & 2
```

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad a = [1, 3] \quad m2 = \begin{bmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{bmatrix}$$

Indexing Arrays

- We can also use ':' to index a range of entries
- E.g all row / columns from a to b by $a : b + 1$

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
m3 = m[:, 1:3 ]          #cols 1 & 2
```

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad m3 = \begin{bmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{bmatrix}$$

- It also works for 1D arrays e.g

```
a = np.linspace(0,4,5)    #a = [0, 1, 2, 3, 4]
a2 = a[1:4]               #a2 = [1, 2, 3]
```

Indexing Arrays

A note on negative indexing

- If you have an array a of length N and you want to index the last element you could write:

```
a = np.linspace(0,4,5)      #a = [0., 1., 2., 3., 4.]
N = len(a)                  #N = 5
s = a[N-1]                  #s = 4.0
s2 = a[N-2]                 #s2 = 3.0
```

- However Python will also accept the shorthand

```
s = a[-1]                   #s = 4.0
s2 = a[-2]                  #s2 = 3.0
```

- This is why if you accidentally index an array by $i - 1$ when $i = 0$ it will give the last element

Indexing Arrays

- The number of rows in a 2D array is given by len(m) or np.size(m,0)
- The number of columns in a 2D array is given by len(m[0]) (i.e. the number of entries in row 0) or np.size(m,1)

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
a = m[:, len(m)-1]                    # a = [3, 6, 9]
s = m[len(m[0])-1, len(m)-2]          # s = 8
```

- Negative indexing also works for 2D arrays:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
a = m[:, -1]          # a = [3, 6, 9]
s = m[-1, -2]         # s = 8
```

2D Arrays and Loops

- To access array entries using loops will require nested loops:
  - one loop to index the row and one to index the column

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
sum = 0
for r in range(0,len(m)):
    for c in range(0,len(m[0])):
        sum = sum + m[r,c]
Avg = sum / m.size      # note Numpy .size attribute
```

- Of course there is also a Numpy function for that:

```
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
Avg = np.sum(m) / m.size
Avg = np.average(m)
```

2D Arrays and Loops

- We can also re-create m using loops
  - one loop to index the row and one to index the column

```
# m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
m = np.zeros ([3, 3],int)
v = 0
for i in range(0,3):
    for j in range(0,3):
        v = v+1
        m[i,j] = v
```

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

2D Arrays and Data Sets

- We can redo the Rainfall example (week 4) using 2D arrays

```
1  import pandas as pd
2  import numpy as np
3
4  # import the Rainfall data
5  Rainfalldata = pd.read_csv("Rainfall.csv")
6  print(Rainfalldata.head())
7
8  # extract a 2D array
9  Rainfall = Rainfalldata[['Rainfall 2010','Rainfall 2020
       ']].values
10 print(type(Rainfall))
11 print(Rainfall)
```

2D Arrays and Data Sets

```
13 #e.g. continued
14
15 # Get the number of rows in the 2D array
16 N_rows = np.size(Rainfall,0)
17
18 # How many regions have seen a decrease in rainfall
19 count = 0
20 for i in range(0, N_rows, 1):
21     if Rainfall[i,1] < Rainfall[i,0]:
22         count = count + 1
23
24 percentage_decreased = count/N_rows*100
25 print('Of the {} regions, {} have reduced rainfall
       which is {:5.2f}% '.format(N_rows, count,
       percentage_decreased))
```