NIHR | Applied Research Collaboration South West Peninsula

@penCHORD_UoE
@peninsula_ARC

Module 1 : Introduction to OR, Data Science and Programming
Recap & Revision Session 2 : Welcome to Python
Elliott Coyne

"Practice Makes Perfect!"

Emirates

HSMA 5

#hsma5isalive

Plan for Session 2

- Any Qs from last week?
- Libraries
- Numpy
- Pandas
- Distributions
- File Access & CSVs *(Brief review as Pandas 'read_csv' does the heavy lifting in many of the use cases)*

Thank you for the feedback – Make sure I break more often!

Lunch 12h00 – 13h00 (Then please consider attending showcase for 30mins)

Warm Up

Answer any of these questions (one or more...)

1. Favourite holiday destination (visited and/ or aspirational)

2. Favourite dish (i.e., food)

3. What ONE item would you take with you to Dan's desert island?

Before we get started...

Any questions from last week?

(Re) Introducing….

Libraries

Libraries and Imports

Libraries are collections of functions that have been written by other people, containing useful bits of code so we don't have to reinvent the wheel every time we write a program. You'll learn about functions in the next session.

To use libraries, we need to *import* them into our program. We usually write import statements at the start of our program.

```
import math # imports the math library
import random # imports the random library
```

The Random Library

Python's "random" library contains a number of functions that allow us to generate random numbers.

This is useful, because we need random numbers if we're going to build *stochastic models.*

Remember our talk about distributions in the very first session…?

Random Library Functions

Now we've imported the Random library, let's look at some of the functions we can use. When we call a function from the library, we need to use the format :

library_name.function_name()

```
import random

# Generate a random number between 0 and 1 sampled from a uniform distribution
random.random()

# Generate a random integer (whole number) between 1 and 10 inclusive
random.randint(1, 10)

my_list = ["Dan", "Mike", "Kerry"]

# Choose a random item from the list my_list
random.choice(my_list)
```

Where to Libraries Live?

See Jupyter Notebook...

Exercise 5 - Are you smarter than Dan as a 4 year old?

When I was 4 years old, I wrote my very first program in BASIC on my Atari 800XL. In it, the computer randomly picked a whole number between 1 and 100, and the user had 10 chances to guess the number. Every time the user guessed a number, they would be told either that the number was “too low”, “too high” or “correct”. If the user used up all 10 chances without guessing correctly, they were told “you lose” and the game would end.

You now have been taught enough to write this program in Python.

You should write the game above (I’d advise using Spyder) along with the following extra features that 4-year old Dan didn’t implement :

- a score, which starts at 1000 and which reduces by 100 for every unsuccessful guess, and which is displayed if the user wins
- the user’s guesses are stored in a list and printed once the game is over
- the game asks if the player wants to play again after every game ends
- after each game, the player’s score is checked against the current high-score (default is 0) and if the last score is higher than the recorded high score then this replaces the high score.

If you do all that with time to spare, add some bells and whistles of your choosing!

You have 45 minutes (+10 minute break), and you should work as a group.

1C

(Re) Introducing….

Numpy

See ‘**session_2a.ipynb**’

NumPy

NumPy is an important Python library for scientific computing.

It provides a number of features, and key amongst them is the ability to easily work with large multi-dimensional arrays **MUCH more efficiently** (computationally).

Because of this, NumPy is particularly important when working in Machine Learning (as you'll see later in the course).

1D

Getting started with NumPy

NumPy is included in many scientific distributions of Python, such as the Anaconda distribution you have installed.

Therefore in order to work with it in our code, we just need to import the library. Convention dictates that we import the library with the alias “np”, so that we refer to NumPy as “np” in the rest of our code :

```
import numpy as np
```

The NumPy Multidimensional Array

The main object in NumPy is the multidimensional array. As with most things in life, things are always better in multiple dimensions!

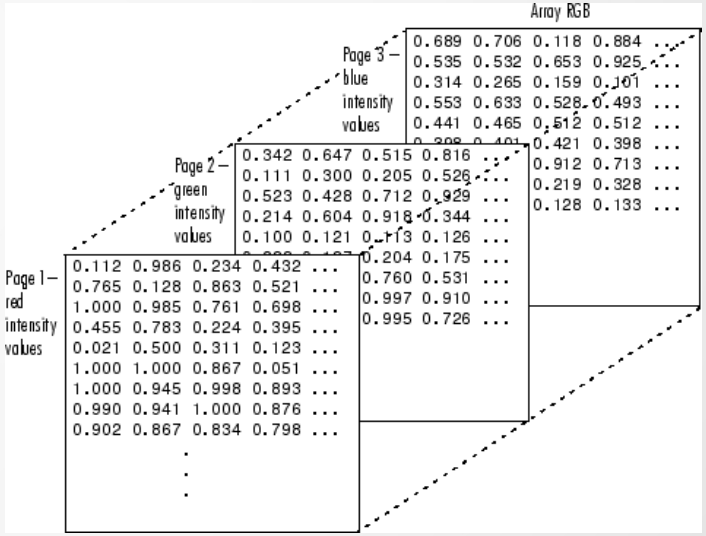
Before we go any further, let's explain what we mean by a multidimensional array.

1D

The NumPy Multidimensional Array

Here's a one dimensional (1D) array :
[1, 2, 3]

Here's a two dimensional (2D) array (an array within an array) :
[[1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]]



Here's a three dimensional (3D) array
(an array within an array within an array) :
[[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
  [[10, 20, 30], [40, 50, 60], [70, 80, 90]],
  [[100, 200, 300], [400, 500, 600], [700, 800, 900]]]

Dimensions in NumPy array are referred to as axes.

1D

The NumPy Multidimensional Array

1D Array Practical Example :

List of ages : 32, 71, 56, 54, 28

[32, 71, 56, 54, 28]

The NumPy Multidimensional Array

2D Array Practical Example :

List of ages for each group :

Group 1 : 32, 71, 56, 54, 28
Group 2 : 17, 28, 22, 18, 56
Group 3 : 98, 88, 10, 12, 5

```
[ [32, 71, 56, 54, 28],
  [17, 28, 22, 18, 56],
  [98, 88, 10, 12, 5] ]
```

1D

The NumPy Multidimensional Array

3D Array Practical Example :

List of ages for each group within each department :

Department 1
Group 1 : 32, 71, 56, 54, 28
Group 2 : 17, 28, 22, 18, 56
Group 3 : 98, 88, 10, 12, 5

Department 2
Group 1 : 8, 9, 15, 16, 23
Group 2 : 21, 37, 45, 42, 46
Group 3 : 51, 67, 16, 16, 21

[[[32, 71, 56, 54, 28], [17, 28, 22, 18, 56], [98, 88, 10, 12, 5]],
[[8, 9, 15, 16, 23], [21, 37, 45, 42, 46], [51, 67, 16, 16, 21]]]

1D

Declaring a NumPy Array

Declaring a NumPy array is easy :

```
import numpy as np

a = np.array([1,2,3,4,5])
```

a is now stored as a NumPy array. In practice, for 2D+ arrays, we often want to read in a series of lists to form a NumPy array. This is easy too :

```
list_1 = [1,2,3,4,5]
list_2 = [2,4,6,8,10]
list_3 = [3,6,9,12,15]

a = np.array([list_1, list_2, list_3])
print(a)
```

```
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]
 [ 3  6  9 12 15]]
```

1D

shape

We can use the *shape* function to find the shape of a NumPy array, in terms of the number of dimensions (if 2 or more), and the number of elements per dimension.

In a 1D array this give us the number of elements in the array :

```
a = np.array([1,2,3])
print (a.shape)
```

```
(3,)
```

e.g. list of three age values

In a 2D+ array this give us the number of rows and columns in multi-dimensional space

```
a = np.array([[1,2,3],[4,5,6]])
print (a.shape)
```

```
(2, 3)
```

e.g. two groups of three age values

```
a = np.array([[[1,2,3],[4,5,6]],
              [[10,20,30],[40,50,60]]])

print (a.shape)
```

```
(2, 2, 3)
```

e.g. two departments with two groups, each with three age values

1D

ndim

We can use the *ndim* function to find out the number of dimensions / axes in our array.

```
c = np.array([[[1,2,3],[4,5,6]],
              [[10,20,30],[40,50,60]]])

print (c.ndim)
```

```
3
```

1D

Homogeneity

NumPy requires arrays to be homegenous. This means that you have to have the same number of columns in each row (in our example before, that would be the same number of age values per group).

This isn't really valid :

```
d = np.array([[1,2,3], [1,2]])
```

but if you wrote the above, you wouldn't get an error. Instead, NumPy will store references to two separate lists within the overall array, and give a shape of (2,) (ie a 1D array with 2 values).

This is not recommended as you will be unable to use a number of key NumPy features if you do this.

Mathematical Operations

NumPy allows us to apply mathematical operations to arrays easily. Let's imagine we have an array a, and we want to double the value of every element in the array :

```
a = np.array([1,2,3,4,5])

a = a * 2
print (a)
```

```
[ 2  4  6  8 10]
```

Slicing

Slicing allows us to easily carve up bits of our NumPy array to deal with only certain sections.

```
b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

# Double the value of every value in the second row only and store in a new
# array
c = b[1] * 2
print (c)

# Print out the value in the third column of the first row
print (b[0][2])
```

```
[12 14 16 18 20]
3
```

1D

Statistics

We can also perform statistical operations on our NumPy array easily. Let's say we want to find the mean of our array :

```
a = np.array([1,2,3,4,5])

# Print the mean of array a
print (a.mean())

b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

# Print the mean across rows (e.g. mean of element 0, mean of element 1 etc)
print (np.mean(b, axis = 0))

# Print the mean across columns (e.g. mean of each row)
print (np.mean(b, axis = 1))
```

```
3.0
[3.5 4.5 5.5 6.5 7.5]
[3. 8.]
```

1D

Dot Product

The *dot product* of two arrays of identical length multiplies the *n*th element of array a with the *n*th element of array b, and adds all of these multiplications together to give a single answer.

Example :

a = [1, 2, 3]
b = [2, 4, 6]

$a \cdot b = (1 \times 2) + (2 \times 4) + (3 \times 6) = 2 + 8 + 18 = 28$

In NumPy we simply use the dot() function :

```
a = np.array([1,2,3])
b = np.array([2,4,6])

dp = np.dot(a, b)
print (dp)
```

```
28
```

Can anyone think of situations where taking the dot product may be useful?

vstack

The *vstack* function allows us to easily add more rows to an existing NumPy array :

```
b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

list_to_add = [3,6,9,12,15]

b = np.vstack([b, list_to_add])

print (b)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [ 3  6  9 12 15]]
```

hstack

The *hstack* function allows us to do the same, but for adding columns :

```
b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

column_to_add = [ [6], [12] ]

b = np.hstack([b, column_to_add])

print(b)
```

```
[[ 1  2  3  4  5  6]
 [ 6  7  8  9 10 12]]
```

Remember – NumPy arrays should be homogenous, and so if you're hstacking new columns, you should be adding new columns to every row in your array.

1D

Removing Duplicate Data

The *unique* function allows us to easily remove duplicate values from an array :

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

# Overwrite b with a version of b with only unique values
b = np.unique(b)

print(b)

b = np.array([ [1,2,3], [4,5,6], [1,2,3] ])

# Remove duplicate rows (axis=0 for rows, 1 for cols, 2+ for 3D+ cols)
b = np.unique(b, axis=0)
print (b)
```

```
[1 2 3 4 5 6 7 8 9]
[[1 2 3]
 [4 5 6]]
```

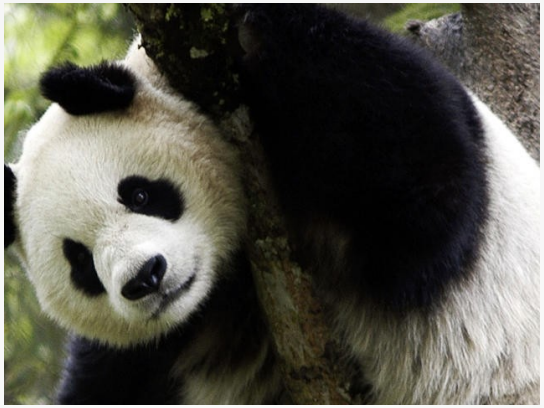
1D

(Re) Introducing….

Pandas

Pandas

Pandas are beautiful creatures native to China



Pandas is also a beautiful open source Python library that provides powerful data structures and analysis tools

The level of geekiness of the person you're talking to can be easily judged by which one of these they assume you're talking about when you say "pandas"

1D

Pandas Overview

Pandas is very powerful for manipulating data in large arrays, and allows for indexing of data.

NumPy and Pandas are often used together, with NumPy used for mathematical functions applied to the data, and Pandas used to manipulate the data.

As with NumPy, there is a conventional alias under which Pandas should be imported :

```
import pandas as pd
```

Pandas DataFrame

One of the most useful structures in Pandas is the Pandas DataFrame. A DataFrame is like a table with different columns (which can have names) for different data fields, and different rows for each entry in the data.

Imagine a table in Excel. Only much more powerful, and with the potential to be multi-dimensional beyond two dimensions.

Patient ID	Name	Age	Location
1	Bob Rogers	46	Plymouth
2	Fred Miles	31	Truro
3	Janet Smith	27	Exeter

1D

Creating a DataFrame

Let's create a new empty dataframe, and set up some lists of data we want to add to the dataframe.

```
import pandas as pd

df = pd.DataFrame()

list_of_patient_IDs = [1,2,3]
list_of_names = ["Bob Rogers", "Fred Miles", "Janet Smith"]
list_of_ages = [46,31,27]
list_of_locations = ["Plymouth", "Truro", "Exeter"]
```

Creating a DataFrame

Now we can setup some columns in our DataFrame by giving the columns names and specifying the data we want in the column (in this case, the data from the lists we created) :

```
df['patient_ID'] = list_of_patient_IDs
df['name'] = list_of_names
df['age'] = list_of_ages
df['location'] = list_of_locations
```

Creating a DataFrame

Now let's print the DataFrame to see what it looks like :

```
print (df)
```

```
   patient_ID          name  age  location
0           1   Bob Rogers   46  Plymouth
1           2   Fred Miles   31     Truro
2           3  Janet Smith   27    Exeter
```

You'll notice that a fifth column has been added automatically – this is the index column. Like a database, a Pandas DataFrame needs a unique identifier for each entry. However, in many cases, a more useful unique identifier will be in our data already (the patient_ID in this case). Let's see how we can make this the index...

Specifying a DataFrame Index

To set the index for the DataFrame, we use the set_index() function. We simply state the name of the column we want to be the index, and set the inplace flag to True (to indicate we want to make a change to an existing DataFrame without creating a new one) :

```
df.set_index('patient_ID', inplace=True)

print (df)
```

```
patient_ID
1             Bob Rogers   46  Plymouth
2             Fred Miles   31     Truro
3            Janet Smith   27    Exeter
```

Retrieving specific records using loc

We can retrieve specific records from our DataFrame using the *loc* method, and specifying the index / indices that we want to retrieve.

```
# Retrieve single entry
# e.g. Print the record with index value of 2
print (df.loc[2])

# Retrieve multiple entries specified in a list
# e.g. Print the records with indices 2 and 3
print (df.loc[[2,3]])

# Retrive multiple entries specified by a slice
# e.g. Print the records with indices 1 to 3 (inclusive)
print (df.loc[1:3])
```

1D

```
name        Fred Miles
age                 31
location         Truro
Name: 2, dtype: object
                  name  age location
patient_ID
2           Fred Miles   31    Truro
3          Janet Smith   27   Exeter
                  name  age  location
patient_ID
1           Bob Rogers   46  Plymouth
2           Fred Miles   31    Truro
3          Janet Smith   27   Exeter
```

Retrieving specific columns

Pandas allows us to easily retrieve specific columns of data (note the index column is automatically included in the output too) :

```
print (df['name'])
```

```
patient_ID
1      Bob Rogers
2      Fred Miles
3     Janet Smith
Name: name, dtype: object
```

Adding more rows (records)

We can add additional rows (records) to a Pandas DataFrame. The most efficient way to do this is to declare a new DataFrame containing the new row(s) to be added, then use the *append* method to join the new DataFrame to the old one. We use a dictionary to establish the new value(s) associated with each column :

```
df2 = pd.DataFrame({"patient_ID":[4,5],
                    "name":["Lucy Rivers", "Dave Jones"],
                    "age":[28,61],
                    "location":["Plymouth", "Plymouth"]})

df2.set_index("patient_ID", inplace=True)

df = df.append(df2)

print (df)
```

patient_ID			
1	Bob Rogers	46	Plymouth
2	Fred Miles	31	Truro
3	Janet Smith	27	Exeter
4	Lucy Rivers	28	Plymouth
5	Dave Jones	61	Plymouth

Adding more columns (fields)

Adding columns to a DataFrame is much easier. We simply set up a list of the values we want to add in the new column, then add the list as a new column to the DataFrame :

```
list_of_previous_admission = [True, True, False, False, False]
df['previously_admitted'] = list_of_previous_admission
print (df)
```

```
patient_ID
1            Bob Rogers   46  Plymouth              True
2            Fred Miles   31     Truro              True
3           Janet Smith   27    Exeter             False
4           Lucy Rivers   28  Plymouth             False
5            Dave Jones   61  Plymouth             False
```

Exercise 5

You have been given a table containing data on Minor Injury Units :

MIU Number	Location	X-Ray Facilities?	Opening Time
463	Lostwithiel	Yes	Morning
571	St Austell	No	Afternoon
614	Truro	Yes	Afternoon
732	Looe	No	Morning
817	Camborne	Yes	Evening

Write code that :
1) Stores the four columns as separate lists (in real applications, you can use functions to read in real data from a .csv file, in case you're worried!)
2) Creates a new Pandas DataFrame that contains this data in named columns, with MIU Number as the index
3) Prints out the data for MIU Numbers 571 and 732
4) Prints out the location column of the data
5) Adds a new column that stores the mean number of patients seen per day, which are 4, 3, 3, 1, 2 respectively, then prints the dataframe.
6) Adds a new row representing the data for MIU number 901, which is a morning MIU with X-Ray facilities in Callington, that sees 5 patients per day, then prints the dataframe.

You have 30 minutes to complete the exercise. Work in your groups.

Further Reading

For the remainder of this session, I want you to read through the tutorial on summary statistics in Pandas here :

https://pandas.pydata.org/docs/getting_started/intro_tutorials/06_calculate_statistics.html

Read through this as a group and check that you all understand how the functions are working.

This tutorial uses the “Titanic” dataset – a commonly used dataset for Machine Learning, which contains information about the passengers on the Titanic. We will use the Titanic dataset later in the course in the Machine Learning module.