Lecture 3 Random Number Generators and Array Methods

What is NumPy -> **Num**erical **Py**thon.

- NumPy provides data types → arrays that facilitate scientific computing with the Python language.
- NumPy provides **functions** that compute on arrays to evaluate mathematical functions
- This week's homework will have you explore some of those functions on your own.
- Todays lecture will focus on a special class of functions

In [1]:

```
import numpy as np
```

Random Number Generators

- Random number generation is a process by which, often by means of a random number **Generator** object, a sequence of numbers or symbols that cannot be reasonably predicted better than by random chance is generated.
- Some of these have existed since ancient times, among whose ranks are well-known "classic" examples, including the rolling of dice or coin flipping, the shuffling of playing cards.
- I am going make use of random numbers here today just to make examples on how to handle arrays, and how to use numpy functions.
- **Importantly, I will show you here how to make random number sequences that you can reproduce**
- Computer generated random numbers are strictly **pseudo-random**

Loading Submodules

- `numpy` has a *lot* of built in functions. More than I know!
- Some of the functions are organized into **submodules** that group together functions of a particular type.
- We can ask python to load a specific submodule, in order to refer to it directly.

The `random` submodule

`from numpy import random`

- In the above line I import the specific **submodule** `random` from `numpy`.
- I am going to call functions or methods from the submodule `random``. If I just import numpy, I would have to refer to a function in random as

`np.random.function_name`

- I want to be able to not type np every time.
- You can always import submodules and specific functions by name to avoid having to write out the module or submodule they come from.

In [2]:

```
from numpy import random
```

Create and *seed* the random number generator

- The first step is to create a random number generator object **rng** using the method `default_rng`.
- In order to control the random number generator we will set the **seed** of the random number generator.
- The **seed** of the random number generator controls the random number sequence generated.
- **If you enter the same seed, you will always get the same sequence of random numbers**
- The **seed** of a random number generator is any integer

In [3]:

```
#set the seed for the random number generator
myseed = 1234
#create the random number generator object
rng = random.default_rng(seed = myseed)
```

Random Integers

- In order to generate integers, we can use the method `integers`. To use this method, we have to specify

```
my_integers = rng.integers(low,high,size)
```

- Here low is the lowest integer (inclusive), high is the highest integers (exclusive)
- size is the dimensions of the numpy array to be created. For a simple array, it is number of random numbers to generate.

- Dice have (usually) 6 faces with the numbers 1 through 6

In [4]:

```
rint1 = rng.integers(1,7,10)
print(rint1)
```

```
[6 6 6 3 2 6 1 2 1 2]
```

- The output is what you might see if you rolled 10 different 6 sided die.

In [5]:

```
rint2 = rng.integers(1,7,10)
print(rint2)
```

```
[4 1 5 2 5 2 5 6 6 2]
```

- Naturally, since these are random numbers, mimicking die rolls, I get different numbers.
- But if we reset the random number seed, we will get the same random numbers.

In [6]:

```
myseed = 1234
rng = random.default_rng(seed = myseed)
rint3 = rng.integers(1,7,10)
print('rint3: ',rint3)
print('rint1: ',rint1)
```

```
rint3:  [6 6 6 3 2 6 1 2 1 2]
rint1:  [6 6 6 3 2 6 1 2 1 2]
```

- The random numbers I get are identical to the first set!
- At first, this may seem disturbing and certainly not at all *random*
- The way to think about it is that there is an infinite sequence of die rolls. The seed selects where in the sequence I start to get numbers.

In [7]:

```
### 5 random numbers between 0 and 9
rdec = rng.integers(0,10,5)
print(rdec)
### Lets not forget integers can be negative. between -10 and 10
rdec2 = rng.integers(-10,11,5)
print(rdec2)
```

```
[5 1 7 2 7]
[-4  6 10 10 -5]
```

Random floating point numbers from a uniform distribution

- In order to generate floating point numbers from a uniform distribution, we can use the method `uniform`. To use this method to create an array **runiform**, we have to specify,

runiform = `rng.uniform` (low,high,size)

- These are floating point numbers (real-valued) ranging low (inclusive) to high (exclusive) but can take any value in between with **equal probability**

In [8]:

```
runiform = rng.uniform(0,1,10)
print(runiform)
```

```
[0.44100612 0.60987081 0.8636213  0.86375767 0.67488131 0.65987435
 0.7357577  0.22275366 0.17206618 0.87041497]
```

- We can increase the range, and incorporate positive and negative numbers

In [9]:

```
runiform2 = rng.uniform(-5,5,10)
print(runiform2)
```

```
[-4.39861342  1.83688909  1.71238019  1.11017981 -4.39862687  4.77769274
 -0.61048373  0.32595022 -4.96867713 -2.48732895]
```

- There are many more ways to generate random numbers.
- The method `normal`, which draws random numbers from a normal distribution, is also useful.

Higher dimensional arrays (Matrices)

- I can also create a higher dimensional array like a matrix, by passing a **tuple** for the dimension of the array.

In [10]:

```
# I want a matrix with 3 rows and 4 columns instead of a vector of 5 numbers, I should provide a tuple (4,3) instead of 5
M = rng.integers(0,10,(3,4))
print(M)
```

```
[[6 8 4 4]
 [1 7 6 9]
 [7 1 8 9]]
```

- To index into a matrix M, I provide a **row** index and **column** index.
- In order to get the entry 2nd row and 3rd column, I have to index M as M[1,2]

In [11]:

```
print(M)
m23 = M[1,2] #entry at row 2, column 3
print('m23: ', m23)
m11 = M[0,0] #first entry of a 3 x 4 matrix
print('m11: ', m11)
m34 = M[2,3] #last entry of a 3 x 4 matrix
print('m34: ',m34)
```

```
[[6 8 4 4]
 [1 7 6 9]
 [7 1 8 9]]
m23:  6
m11:  6
m34:  9
```

- Of course, if I go out of range, python will just complain.

In [12]:

```
M[3,4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[12], line 1
----> 1 M[3,4]

IndexError: index 3 is out of bounds for axis 0 with size 3
```

- We can address a single row or column of the matrix by

In [13]:

```python
print(M)
M2 = M[1]
print('M2: ', M2)
```

```
[[6 8 4 4]
 [1 7 6 9]
 [7 1 8 9]]
M2:  [1 7 6 9]
```

- In my opinion, the above is very poor syntax because it is ambiguous which axis you are referring to.
- It requires that you know that python defaults to rows, and that a single index will extract that row.
- The same thing can be achieved with more clarity using `:` to indicate *ALL ELEMENTS*

In [14]:

```
R2 = M[1,:] # row 2
C1 = M[:,0] # column 1
print('M:', M)
print('R2: ',R2)
print('C1: ',C1)
```

```
M: [[6 8 4 4]
 [1 7 6 9]
 [7 1 8 9]]
R2:  [1 7 6 9]
C1:  [6 1 7]
```

- I can extract a row and column of a matrix into a new array and do some math on it

In [15]:

```
m = M[:,0]  # I grabbed the first column of M and copied it into m
u = m**2 - 5 # I squared the values and subtracted v and placed in a new array u
print(u)
```

```
[31 -4 44]
```

- I can manipulate a specific row or column in the matrix

In [16]:

```
print(M)
M[:,0] = u #replace the first column with u
M[:,1] = -10 #replace the second column with -10 at all locations
print(M)
```

```
[[6 8 4 4]
 [1 7 6 9]
 [7 1 8 9]]
[[ 31 -10   4   4]
 [ -4 -10   6   9]
 [ 44 -10   8   9]]
```

Size matters!

In [17]:

```
### Each row of M has 4 elements. If I try to place a vector length other than 4 into a row of M it will fail
v = np.array([1,2,3])
M[0,:] = v
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[17], line 3
      1 ### Each row of M has 4 elements. If I try to place a vector length other than 4 into a row
of M it will fail
      2 v = np.array([1,2,3])
----> 3 M[0,:] = v

ValueError: could not broadcast input array from shape (3,) into shape (4,)
```

In [ ]:

```
## IF the size matches, I will succeed.
v = np.array([1,2,3,4])
M[0,:] = v
```

The general concept of **axis**

- I referred to the rows and columns of a matrix. Python thinks of these as the **axis** of the array.

1. axis 0 is the rows
2. axis 1 is the columns
3. axis 2 is the ???? ...

- There is no limit to the number of dimensions to a python array. Right now we are working with
    - vector (1 dimensional array)
    - matrix (2 dimensional array).
- But, in many practical applications we need more than 2 dimensions.
- For example, when we record brain images of blood flow from human subject, each image has 3 dimensions, and there is a 4th dimension of time.

Maximum and minimum

There are three pairs of functions that handle maximum and minimum of arrays.

1. Within an array to find the maximum/minimum

- `max`
- `min`

2. to find the *index* of the maximum or minimum element of an array

- `argmax`
- `argmin`

3. to compare two equal size arrays element by element, use

- `maximum`
- `minimum`

- Lets get the maximum and minimum of an array and the index of the maximum and minimum of an array

In [18]:

```
v = rng.integers(1,21,15) #15 integers ranging from 1 to 20
maxv = np.max(v) # find the maximum value of v
index_maxv = np.argmax(v) #find the index into v that gives the maximum value.
print('v = ',v)
print('maxv = ',maxv)
print('index_maxv =', index_maxv)
minv = np.min(v) #find the minimum value of v
index_minv = np.argmin(v) # find the index into v that gives the minimum value
print('v = ',v)
print('minv = ',minv)
print('index_minv =', index_minv)
```

```
v =  [ 8  4 18 19 20  2 18 10  9 17 19  6 16 18 18]
maxv =  20
index_maxv = 4
v =  [ 8  4 18 19 20  2 18 10  9 17 19  6 16 18 18]
minv =  2
index_minv = 5
```

- When working with a matrix we often want to compute the maximum or minimum of each row or column. To do that, we have to specify an *axis*

In [19]:

```
M = rng.integers(1,21,(7,3)) # random numbers between 1 and 20 in a 5 x 6 matrix
maxM_0 = np.max(M,axis = 0)
maxM_1 = np.max(M,axis = 1)
print('M')
print(M)
print('max, axis = 0') # maximum of each column,
print(maxM_0)
print('max, axis = 1') # maximum of each row,
print(maxM_1)
```

```
M
[[20  7 17]
 [ 9  9 16]
 [ 8 16 10]
 [ 1  2 16]
 [ 5 20 11]
 [ 4 15  4]
 [ 9 12  7]]
max, axis = 0
[20 20 17]
max, axis = 1
[20 16 16 16 20 15 12]
```

I can also compare two arrays and choose the maximum or minimum element by element

In [20]:

```
w = rng.integers(1,7,10)
u = rng.integers(1,7,10)
print('w = ',w)
print('u = ',u)
p = np.maximum(u,w)
q = np.minimum(u,w)
print('p = ',p)
print('q = ',q)
```

```
w =  [5 6 3 5 2 3 1 3 5 5]
u =  [2 6 4 2 2 5 4 5 5 6]
p =  [5 6 4 5 2 5 4 5 5 6]
q =  [2 6 3 2 2 3 1 3 5 5]
```

Sort functions for `numpy` arrays

- In many operations with data it is useful to be able to sort the data from lowest to highest, or highest to lowest.
- It is perhaps not surprising that the function that will sort an array is called `sort`

In [21]:

```
v = rng.integers(1,7,10)  # 10 random numbers between 1 and 6
v_sorted = np.sort(v) #sort v in ascending order
print('v =', v)
print('v_sorted =',v_sorted)
```

```
v = [2 1 3 4 5 4 5 2 1 3]
v_sorted = [1 1 2 2 3 3 4 4 5 5]
```

- Numpy `sort` function always sorts in ascending order from lowest to highest. What if I wanted to sort from highest to lowest?
- Numpy has a `flip` function that allows up reverse the order of the elements in an array.

In [22]:

```
v_flipped = np.flip(v)
print('v = ',v)
print('v_flipped = ', v_flipped)
```

```
v =  [2 1 3 4 5 4 5 2 1 3]
v_flipped =  [3 1 2 5 4 5 4 3 1 2]
```

In [23]:

```
v_sorted = np.sort(v)
v_sorted = np.flip(v_sorted)
print('v_sorted =', v_sorted)
```

```
v_sorted = [5 5 4 4 3 3 2 2 1 1]
```

In [24]:

```
#I could actually do it one step by **nesting** my functions like this.
v_sorted = np.flip(np.sort(v)) # I implicitly take the output of np.sort and enter into np.flip
print('v_sorted =', v_sorted)
```

```
v_sorted = [5 5 4 4 3 3 2 2 1 1]
```

Ordered Indices - `argsort`

- In many (*most?*) circumstances you don't only want to be able to obtain a sorted list of items, but you also want to know *what order of indices* produces the sorted list. This may not seem obvious, but i will make some examples here that illustrate why this is important.
- The `argsort` function tells you the order of indices to sort an array

In [25]:

```
v = rng.integers(1,7,10)
v_sorted = np.sort(v) #This obtains a sorted list in increasing order.
sort_order = np.argsort(v) #This obtains a list of ordered indices that you could use to sort v
v_sorted_byorder = v[sort_order]
print('i = ',np.arange(0,10,1)) # i juat wanted to track the index
print('v = ',v)
print('sort_order = ',sort_order)
print('v_sorted = ',v_sorted)
print('v_sorted_byorder = ', v_sorted_byorder)
```

```
i =  [0 1 2 3 4 5 6 7 8 9]
v =  [2 4 4 6 5 1 4 2 3 2]
sort_order =  [5 0 7 9 8 2 6 1 4 3]
v_sorted =  [1 2 2 2 3 4 4 4 5 6]
v_sorted_byorder =  [1 2 2 2 3 4 4 4 5 6]
```

- Why is this useful?
- Many times, we want to sort data on one variable, *and sort other variables in the same order*
- I provide an example here on the relationship between age and LDL-bad cholesterol.

In [26]:

```
age = np.array([55,58,72,46,48,65]) #age in years
LDL = np.array([65,90,120,55,70,100]) #LDL - bad cholesterol
```

- I want to quickly look at those numbers and determine if LDL goes up with age.
- What I'm going to do is sort the data by age and then use that sort order with the LDL data.

In [27]:

```
age_order = np.argsort(age)
age_sorted = age[age_order]
LDL_sorted_byage = LDL[age_order] # notice i passed the indices to order age into LDL
print('age = ', age_sorted)
print('LDL = ', LDL_sorted_byage)
```

```
age =  [46 48 55 58 65 72]
LDL =  [ 55  70  65  90 100 120]
```

Mathematical Functions in Python

All of the mathematical functions such as logarithms, exponentials, trignometric functions, etc. can be found in numpy

- A good first guess is usually `np.exp`, `np.log`, `np.sin`, `np.cos`, 'np.sin`, etc.
- Special numbers like π are obviously `np.pi`

How do I look stuff up

Google it!

*An important point to always keep in mind is that numpy functions work on arrays!

In [28]:

```
x = np.array([1,10,100,1000])
y = np.log10(x) # this is the base 10 logarithm
print('x = ',x)
print('y = ',y)
```

```
x =  [   1   10  100 1000]
y =  [0. 1. 2. 3.]
```

In [29]:

```
z = np.array([0.1,0.01,0.001])
a = np.log10(z)
print(z)
print(a)
```

```
[0.1   0.01  0.001]
[-1. -2. -3.]
```

Think in terms of arrays when possible.