UECM1703 Introduction to Scientific Computing
Topic 2: Arrays Manipulation

Lecturer: Dr Liew How Hui (liewhh@utar.edu.my)

Oct 2022

Programming with basic data types and imperative programming is not convinient and efficient for numeric array data.

The Numpy module gains popularity because it provides many important array operations such as the array slicing (Section 2.2), the basic arithmetic, logical and relational operations for arrays (Section 2.3). Array reduction functions are investigated in Section 2.4. Finally, in Section 2.7, we investigate a bit about the I/O of arrays.

CO1: perform vector and matrix operation using computer software

Contents

§2.1 Numpy Array Data Types

An ***array*** is a **multi-index** and **homogeneous** (elements is of the same type) data structure. The array of floating point numbers, array of integers and array of Booleans are fundamental.

Python's Numpy array `numpy.ndarray` is an n-**dimensional** (determines the **number of indices**, abbreviated as n-D) array object. Note that an n-D array is called **tensor** in the tensorflow machine learning package.

To use Python's Numpy array, be sure to import the library as follows.

```
import numpy as np
```

§2.1.1 Numpy Array Construction

Example 2.1.1. Various ways to create a 1-D array:

- Constructing an array with no particular pattern:

```
# A1 is a 1-D array of integers
A1 = np.array([7,19,19,18])
# A2 is a 1-D array of doubles
A2 = np.array([7,19,19,18], dtype='double')
```

- Constructing an array with a particular pattern — with 0, 1 or some constant.

```
A3 = np.zeros(10)        # 10 zeros of data type double
A4 = np.ones(10)         # 10 ones of data type double
A5 = np.full(10,100)     # 10 hundreds of data type integer
A6 = np.linspace(0,np.pi,num=51,endpoint=True)
```

The command `np.linspace` is usually used in the creation of a 1-D array for the interval of a particular function $f(x)$. In particular,

$$\texttt{A6} = 0,\ \frac{\pi}{50},\ \frac{2\pi}{50},\ ...,\ \frac{49\pi}{50},\ \pi.$$

Note that interval $[0, \pi]$ is cut into **50 intervals** with **51 points**.

- Constructing an array with a particular pattern — arithmetic sequence:

```
# For an array of some order, we can use np.arange
A7 = np.arange(5)       # 0, 1, 2, 3, 4
A8 = np.arange(1,5)     # 1, 2, 3, 4
A9 = np.arange(1,50,5) # 1,  6, 11, 16, 21, 26, 31, 36, 41, 46
```

- Constructing an array with a particular pattern — function sequence:

```
# Sine sequence.  Make sure sin is import from math
A10 = np.array([sin(0.1), sin(0.2), sin(0.3), sin(0.4)])
A11 = np.array([sin(x) for x in np.arange(0.1,0.5,0.1)])
```

- Constructing an array with random numbers:

```
# To prevent the random numbers to be different every time
# we generate them, we can set the seed:
np.random.seed(123)            # any integer can be used as seed
A12 = np.random.rand(50)       # 50 random numbers ~ Uniform[0,1)
A13 = np.random.randn(50)      # 50 random numbers ~ Normal(0,1)
```

Example 2.1.2. Various ways to create a 2-D array:

- Constructing an array with no particular pattern:

```
# B1 is a 2-D array of integers with first row [7, 19] and
# second row [19, 18]
B1 = np.array([[7,19],[19,18]])
# A2 is a 2-D array of doubles
B2 = np.array([[7,19],[19,18]], dtype='double')
```

- Constructing an array with a particular pattern — with 0, 1 or some constant.

```
# We are using tuples to construct 2D array
B3 = np.zeros((2,4))
B4 = np.ones((4,2))
B5 = np.full((3,5),100)
```

$$\texttt{B3} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \texttt{B4} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \texttt{B5} = \begin{bmatrix} 100 & 100 & 100 & 100 & 100 \\ 100 & 100 & 100 & 100 & 100 \\ 100 & 100 & 100 & 100 & 100 \end{bmatrix}$$

- Constructing an array with particular patterns — diagonal matrices and the variations.

```
# We are using tuples to construct 2D array
B6  = np.eye(4)         # 4 x 4 identity matrix
B7  = np.eye(3,2)       # 3 x 2 identity matrix(?)
B8  = np.diag([5,7,-3,4])   # Create a diagonal matrix
B9  = np.diag(np.arange(6,2,-1))
B10a, B10b = np.meshgrid([1,2,3],[2,5,7,9]) # return 4x3 matrices
# default indexing = 'xy' for computer graphics (Topic 4)
```

$$\texttt{B6} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \texttt{B7} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad \texttt{B8} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & -3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \qquad \texttt{B9} = \begin{bmatrix} 6 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

$$\texttt{B10a} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad \texttt{B10b} = \begin{bmatrix} 2 & 2 & 2 \\ 5 & 5 & 5 \\ 7 & 7 & 7 \\ 9 & 9 & 9 \end{bmatrix}$$

- Constructing an array with a particular pattern — a matrix of function values:

```
from math import tan, sin, cos, pi, log, exp
B11 = np.array([[tan(pi/3), 3/sin(pi/4)],
                [log(cos(pi/6)), 1+exp(1.5)]])
B12 = np.array([sin(x) for x in np.linspace(0,5*pi/14,6)]).reshape(
      (2,3))
B13 = np.vander([1,3,2,5])
B14 = B13.diagonal()        # Get the diagonal of a matrix!
B15 = np.diag(B13)
```

$$\texttt{B11} = \begin{bmatrix} \tan\frac{\pi}{3} & \frac{3}{\sin(\pi/4)} \\ \ln\cos\frac{\pi}{6} & 1+e^{1.5} \end{bmatrix} \qquad \texttt{B12} = \begin{bmatrix} \sin 0 & \sin\frac{\pi}{14} & \sin\frac{2\pi}{14} \\ \sin\frac{3\pi}{14} & \sin\frac{4\pi}{14} & \sin\frac{5\pi}{14} \end{bmatrix} \qquad \texttt{B13} = \begin{bmatrix} 1^3 & 1^2 & 1^1 & 1^0 \\ 3^3 & 3^2 & 3^1 & 3^0 \\ 2^3 & 2^2 & 2^1 & 2^0 \\ 5^3 & 5^2 & 5^1 & 5^0 \end{bmatrix}$$

- Constructing an array with random numbers:

```
B16 = np.random.rand(3, 2)       # 3x2 random matrix uniform over [0,1)
B17 = np.random.random((3,2))    # 3x2 random matrix uniform over [0,1)
B18 = np.random.randn(3, 2)      # 3x2 random matrix Normal(0,1)
```

Example 2.1.3. Various ways to create a 3-D array:

- Constructing an array with no particular pattern:

```
C1 = np.array([[[1,2],[1,4],[5,1]],[[7,2],[9,3],[8,8]]])
```

$$C1[0,:,:] = \begin{bmatrix} 1 & 2 \\ 1 & 4 \\ 5 & 1 \end{bmatrix}, \quad C1[1,:,:] = \begin{bmatrix} 7 & 2 \\ 9 & 3 \\ 8 & 8 \end{bmatrix}$$

- Constructing an array with a particular pattern — with 0, 1 or some constant.

```
# We are using tuples to construct 3D arrays
C3 = np.zeros((2,4,3))
C4 = np.ones((4,2,3))
C5 = np.full((3,5,3),100)
```

$$C4[0,:,:] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, C4[1,:,:] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, C4[2,:,:] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, C4[3,:,:] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

- Constructing an array with particular patterns — 3-D grid.

```
C6, C7, C8 = np.meshgrid([1,2],[2,5,7],[8,9],indexing='ij')
  C6               C7                C8
  |                |                 |
  V                V                 V
[[[1  1]         [[[2  2]           [[[8  9]
  [1  1]           [5  5]             [8  9]
  [1  1]]          [7  7]]            [8  9]]

 [[2  2]          [[2  2]            [[8  9]
  [2  2]           [5  5]             [8  9]
  [2  2]]]         [7  7]]]           [8  9]]]
```

- Constructing an array with random numbers:

```
C13 = np.random.rand(3,2,4)      # 3x2x4 random matrix ~ Uniform[0,1)
C14 = np.random.random((3,2,4)) # 3x2x4 random matrix ~ Uniform[0,1)
C15 = np.random.randn(3,2,4)     # 3x2x4 random matrix ~ Normal(0,1)
```

- Construct from a coloured image — in Python's ImageIO module, the class `Image` is a subclass of `np.array`.

Relation of real-world data to arrays:

- Audio / Sound Signals — 1-D arrays
- Black-and-White images, grayscale images — 2-D arrays
- Colour images — 3-D arrays (RGB colour = $M \times N \times 3$ with values in [0,1] or 0–255; RGBA colour = $M \times N \times 4$ with values in [0,1] or 0–255 with A=transparency; Out-of-range RGB(A) values are clipped.)

Python image visualisation tools:

```
matplotlib.pyplot.spy(Z)   # 2D array only
matplotlib.pyplot.imshow(X, cmap=None, norm=None, *, aspect=None,
  interpolation=None, alpha=None, vmin=None, vmax=None, origin=None,
  extent=None, interpolation_stage=None, filternorm=True,
  filterrad=4.0, resample=None, url=None, data=None, **kwargs)
```

For grayscale images, use cmap='gray', vmin=0, vmax=255.

§2.1.2 Basic Array Attributes (Shape, Size)

Arrays are created with a particular shape and data type. The information or attributions associated with arrays can be obtained from the Numpy array.

- Get the dimension of an array: `A.ndim`.

  The dimensions of the matrices of in Examples 2.1.1, 2.1.2 and 2.1.3 are 1, 2 and 3 respectively.

- Get the shape of an array: `A.shape`.
- Get the number of elements in an array: `A.size`.
- Get the total number of bytes used: `A.nbytes`, it is defined as `A.size*A.itemsize`. For example, if there are n elements in `A` and all the elements are 64-bit floating numbers (Topic 1), then the total number of bytes used in `A` to store array data is $8n$.

There are a few operations which are related to the attributes of an array:

- Transpose: It works by reversing the 'indices' but for real-world application, it is used to transpose a matrix `A` to `A.T`, for example:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \xrightarrow{T} \begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \end{bmatrix}$$

- Change from n-D array to 1-D array: `A.ravel()`, `A.flatten()`
- Change a shape to a compatible shape: `A.reshape(`$(d_1, d_2, ..., d_r)$`)`
- Change a shape to a **smaller** shape: `A.resize(`$(d_1, d_2, ..., d_r)$`)`

§2.1.3 Numpy Array Formatting

- Get: `get_printoptions(`*`args`*`)`
- Set: `set_printoptions(`*`args`*`)`
  - *`args`* are `precision` (default 8), `threshold` (default 1000), `edgeitems` (default 3), `linewidth` (default 75 characters per line), `suppress` (default: False, i.e. do not print small floating point values using scientific notation), `nanstr`, `infstr`, `formatter` (takes a dictionary), `sign`, `floatmode`.
- Print: `print(A)`

Example 2.1.4. Let's see how 'printoptions' is used to control the printing of array.

```
A = np.array([[1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9],
              [1/8, 1/9, 1/8, 1/7, 1/6, 1/5, 1/4, 1/3]])
print(A)
default_printoptions = np.get_printoptions()
np.set_printoptions(precision=5, linewidth=100)
print(A)
np.set_printoptions(**default_printoptions)
print(A)
```

§2.2 Array Indexing

Indexing is an important way to assess the data in an n-D array. Indexing an array with an integer / integers will lead to the reduction of dimension by default. To keep the dimension, we either use a range `m:n:s` or use an 'extra' index called 'None' to keep the dimension.

We need to note that indexing an array `A` on gives us a **view** of the array `A`.

- Return a "view" of `A` with a given shape $(m_1, \cdots, m_k)$. Note that $m_1 \times \cdots \times m_k$ must be equal to `A.size`: `A.reshape((`m_1`,...,`m_k`))`.
- Return a "view" of $(m_1, m_2, \cdots, m_k)$-array `A` as a transpose with a shape $(m_k, \cdots, m_2, m_1)$: `A.T` (alternatively, `np.transpose(A)`).

Making any changes to the **view** will be reflected on the original array. If we need a **copy** of the sub-array from `A`, we need to use the '`.copy()`' method or stacking commands:

- Return a "copy" of `A` with a specific type: `A.astype(sometype)`, here `sometype` can be `'double'`, `'bool'`, `'int8'`, etc.
- Return a new array by stacking existing array(s): `np.hstack` and `np.vstack`.

$$\texttt{np.hstack}((A_1, A_2, ..., A_k)) = [A_1\ A_2\ ...\ A_k], \quad \texttt{np.vstack}((A_1, A_2, ..., A_k)) = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{bmatrix}.$$

§2.2.1 Usual Indexing

Attention 1: Python's index starts from 0

Attention 2: Python's ending index `m:n` will never reach `n`

Without loss of generality, we consider a 2-D array `A`.

(I) Indexing an element of `A` at `(m,n)`: `A[m-1,n-1]` ..... dim=0

(II) Indexing `A` at `m`-row: `A[m-1,:]` ..... dim=1

(III) Indexing `A` at `n`-column: `A[:,n-1]` ..... dim=1

(IV) Indexing a sub-array of `A` bounded by `m1`-row to `m2`-row and `n1`-column to `n2`-column: `A[m1-1:m2,n1-1:n2]`

(V) Indexing a sub-array of `A` from `m1`-row to row-`m2` by step `ms` and from `n1`-column to `n2`-column by step `ns`: `A[m1-1:m2:ms,n1-1:n2:ns]`. When `m1-1` is ignored, it is assumed to be 0 and when `m2` is ignore, it is assumed last row and when `ms` is ignored, it is assumed 1. This is similar for `n1-1`, `ns` and `n2`.

(VI) Indexing a sub-array of `A` using rows `r1`, $\cdots$, `rk`, columns `c1`, $\cdots$, `cl`: `A[[r1-1,...,rk-1],:][:,[c1-1,...,cl-1]]`

Remark 2.2.1. Numpy provides an alternative indexing function `take` (or `put`) to take (or assign values to) a "slice" of elements from an array `A`. To get an element from an item, the function `item` is used instead:

```
(a) A.item(3,4)
(b) A.take(indices=[3],axis=0)
(c) A.take(indices=[4],axis=1)
(d) A.take(range(3,7),axis=1).take(range(1,3),axis=0)
(d) A.take([3,5,7],axis=1).take([1,3,5],axis=0)
```

Example 2.2.2. Consider the following array:

A =

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54

which is generated with `A = np.arange(1,55,dtype='double').reshape(6,9)`. What is the output of the following commands:

(a) `print(A[3,4])` Indexing (I)
(b) `print(A[3,:])` Indexing (II)
(c) `print(A[:,4])` Indexing (III)
(d) `print(A[1:3,3:7])` Indexing (IV)
(e) `print(A[1:6:2,:][:,3:8:2])` Indexing (V) & (VI)
(f) `print(A[3,None])` (It is the same as `A[None,3]`)
(g) `print(A[:,4,None])` or `print(A[:,4:5])`

Solution:

(a) `32.0`
(b)
```
[28.  29.  30.  31.  32.  33.  34.  35.  36.]
```
(c)
```
[ 5.  14.  23.  32.  41.  50.]
```
(d)
```
[[13.  14.  15.  16.]
 [22.  23.  24.  25.]]
```
(e)
```
[[13.  15.  17.]
 [31.  33.  35.]
 [49.  51.  53.]]
```
(f)
```
[[28.  29.  30.  31.  32.  33.  34.  35.  36.]]
```
(different from part (b) in terms of dimension, this is a 1x9 'matrix')
(g)
```
[[ 5],[14],[23],[32],[41],[50]]
```
(we obtain a 6x1 column 'matrix' rather than a 1-D array as in part (c))

Example 2.2.3 (Final Exam Sept 2015, Q3(a))**.** The following vector is defined in Python

```
V = np.array([2,7,-3,5,0,14,-1,10,-6,8])
```

What will be displayed if the following variables `B`, `C` and `D` are printed.

(i) `B = V[[1,3,4,5,6,9]]` (2 marks)

```
      0  1  2  3  4  5  6  7  8  9
V =   2  7 -3  5  0 14 -1 10 -6  8
```

Solution: `B = [7 5 0 14 -1 8]`

(ii) `C = V[[8,2,1,9]]` (2 marks)

Solution: `C = [-6 -3 7 8]`

(iii) `D = np.array([V[[0,2,4]],V[[1,3,5]],V[[2,5,8]]])` (2 marks)

Solution: `D = [[2, -3, 0], [7, 5, 14], [-3, 14, -6]]`

Example 2.2.4 (Final Exam Sept 2015, Q1(a))**.** The following matrix is defined in Python:

$$M = \begin{bmatrix} 6 & 9 & 12 & 4 & 3 & 0 \\ 4 & 4 & 15 & 2 & 1 & 1 \\ 2 & 1 & 18 & -5 & 8 & 2 \\ -6 & -4 & 21 & 1 & -5 & 2 \end{bmatrix}.$$

What will be displayed if the following variables `A` in (i) to `C` in (iii) are printed?

(i) `A = M[[0,2],:][:,[1,3]]` (2 marks)
(ii) `B = M[:,[0,3,4,5]]` (2 marks)
(iii) `C = M[1:3,:]` (2 marks)

One has to be very careful with the indexing in Numpy because it only create **views** and does not create **copies**.

Example 2.2.5 (View vs Copy of a sub-array)**.** Study the following Python instructions and explain what is each output:

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
B = A[:,0:2]
B[0,0] = 8
# We only change an element in B
print("A=",A)
print("B=",B)
```

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
B = A[:,0:2].copy()
B[0,0] = 8
print("A=",A)
print("B=",B)
```

Example 2.2.6 (3-D array)**.** For the 3-D array

$$A[0,:,:] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad A[1,:,:] = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}.$$

write down the output of the following commands.

(a) `print(A[0,2,1])`

(b) `print(A[:,2,1])`

(c) `print(A[0,:,1])`

(d) `print(A[:,2,:])`

(e) `print(A[0:1,1:3,1:2])`

(f) `print(A[[1,0],:,:][:,[2,1,0],:])`

Example 2.2.7 (Combining Programming and Array Indexing. A Difficult Example)**.** Generate an $n \times n$ clockwise spiral matrix using Python. E.g.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 15 & 16 & 17 & 18 & 5 \\ 14 & 23 & 24 & 19 & 6 \\ 13 & 22 & 21 & 20 & 7 \\ 12 & 11 & 10 & 9 & 8 \end{bmatrix}.$$

Sample Solution inspired by `https://rosettacode.org/wiki/Spiral_matrix`

```
def spiral(n,m=None):
    _n,_m = (n,m) if m is not None else (n,n)
    _nl,_ml=0,0
    dx,dy = 0,1            # Starting increments
    x,y = 0,0              # Starting location
    import numpy as np
    myarray = np.zeros((_n,_m),dtype='int')
    for i in range(_n*_m):
        myarray[x,y] = i
        nx,ny = x+dx, y+dy # (dx,dy) = direction to update array
        if _nl<=nx<_n and _ml<=ny<_m:
            x,y = nx,ny
        else:
            if dx==0 and dy==1:
                _nl+=1; dx,dy=1,0
            elif dx==1 and dy==0:
                _m-=1; dx,dy=0,-1
            elif dx==0 and dy==-1:
                _n-=1; dx,dy=-1,0
            elif dx==-1 and dy== 0:
                _ml+=1; dx,dy=0,1
            else:
                return None # Should not reach this state
            x,y = x+dx,y+dy
    return myarray

print(spiral(5))
```

Example 2.2.8 (Combining Programming and Array Indexing in Solving Differential Equations Numerically)**.** In applying finite difference approximation to the type 1 ODE-BVP:

$$y''(x) + p(x)y'(x) + q(x)y(x) = r(x), \quad a < x < b, \quad y(a) = y_a,\ y(b) = y_b$$

we obtain matrix

$$C = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 - \frac{h}{2}p(x_1) & (h^2 q(x_1) - 2) & 1 + \frac{h}{2}p(x_1) & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 1 - \frac{h}{2}p(x_{n-1}) & (h^2 q(x_{n-1}) - 2) & 1 + \frac{h}{2}p(x_{n-1}) \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

where $h = \frac{b-a}{n}$. Write a Python script to generate the matrix C.

Example 2.2.9 (Combining Programming and Array Indexing in Technical Analysis of Financial Data)**.** In the past, it is possible for us to download a lot of stock price data from Yahoo!Finance. However, Yahoo!Finance is providing less and less stock price data for data and is transforming itself to pay-per-use service. The stock price data of Telekom Malaysia (a listed company which provides the most expensive broadband service in South East Asia) below was downloaded a few years ago.

```
Date,Open,High,Low,Close,Volume,Adj Close
2016-12-30,6.06,6.09,5.81,5.95,5842300,5.95
2016-12-29,6.05,6.12,5.98,6.06,6777900,6.06
2016-12-28,5.96,6.06,5.96,6.03,2503700,6.03
2016-12-27,5.93,5.99,5.92,5.99,922400,5.99
2016-12-26,5.95,5.95,5.95,5.95,000,5.95
2016-12-23,5.91,5.97,5.91,5.95,838100,5.95
2016-12-22,5.97,5.97,5.92,5.95,1065000,5.95
2016-12-21,6.02,6.02,5.92,5.95,3405900,5.95
2016-12-20,5.95,6.07,5.94,5.98,3101400,5.98
2016-12-19,5.97,6.00,5.91,5.95,2006000,5.95
2016-12-16,5.90,5.96,5.89,5.95,3975600,5.95
2016-12-15,5.91,5.95,5.90,5.90,3987400,5.90
2016-12-14,5.96,6.00,5.93,5.95,5128000,5.95
2016-12-13,6.04,6.05,5.94,5.96,4111000,5.96
2016-12-12,6.03,6.03,6.03,6.03,000,6.03
2016-12-09,6.11,6.11,6.01,6.03,1573000,6.03
2016-12-08,6.16,6.20,6.11,6.11,3189800,6.11
2016-12-07,6.13,6.15,6.09,6.11,4564800,6.11
2016-12-06,6.12,6.15,6.09,6.12,2976600,6.12
2016-12-05,6.15,6.19,6.13,6.14,3303800,6.14
2016-12-02,6.15,6.22,6.09,6.13,2134000,6.13
2016-12-01,6.17,6.24,6.14,6.15,6188400,6.15
```

By using the closing price, write a Python program to calculate

- the price difference between the next day and today for December 2016;
- the three-day (moving) average for December 2016. [Useful reference: `https://rosettacode.org/wiki/Averages/Simple_moving_average`]

Sample Solution

```
1 import numpy as np
```

```
dclose = np.array([6.15, 6.13, 6.14, 6.12, 6.11, 6.11, 6.03,
                   6.03, 5.96, 5.95, 5.9, 5.95, 5.95, 5.98,
                   5.95, 5.95, 5.95, 5.95, 5.99, 6.03, 6.06, 5.95])
price_diff = np.zeros(dclose.size-1)
moving3 = np.zeros(dclose.size-2)
for today in range(price_diff.size):
    next_day = today + 1
    price_diff[today] = dclose[next_day]-dclose[today]
for today in range(moving3.size):
    next_day = today + 1
    next_2day = today + 2
    moving3[today] = (dclose[today]+dclose[next_day]+dclose[next_2day])/3
print("Price difference between next day and today for December 2016: ")
print(price_diff)
print("3-D moving average for December 2016:", moving3)
```

§2.2.2 Negative Indexing

Indexing array with the usual bounded index is often enough but Python provides **"Negative" indexing** which is used to index an array "from last element". For example, `-1` refers to the last index, `-n` refers to the `n` last index. Note that going beyond index bound can cause error!

Example 2.2.10. Consider the following integer array:

`A` =

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54

What is the output of the following commands:

(a) `print(A[-2,4])`
(b) `print(A[-2,:])`
(c) `print(A[:,-4])`
(d) `print(A[1:-3,3:-4])`
(e) `print(A[1:-1:2,:][:,2:-2:2])`
(f) `print(A[-2,None])`
(g) `print(A[:,-4:-5:-1])`

Solution:

(a) `41` (second last row, fifth column)
(b) `[37, 38, 39, 40, 41, 42, 43, 44, 45]` (second last row)
(c) `[ 6, 15, 24, 33, 42, 51]` (fourth last column = sixth column)
(d) `[[13 14] [22 23]]` (second row to the row before the last third row **and** fourth column to before the last fourth column)
(e) `[[12 14 16] [30 32 34]]` (second and fourth rows **and** third, fifth and seventh columns)
(f) `[[37 38 39 40 41 42 43 44 45]]` (different from part (b) in terms of dimension, this is a 1x9 'matrix')
(g) `[[ 6] [15] [24] [33] [42] [51]]` (we obtain a 6x1 column 'matrix' rather than a 1-D array as in part (c))

§2.2.3 Array Mathematical Functions and Numpy Ufuncs

When we encounter an array below:

```
A = np.array([sin(1), sin(2), sin(3), sin(4), sin(5), sin(6)])
```

we would hope to **abbreaviate** it as

```
A = sin(np.array([1,2,3,4,5,6]))
```

However, the sine function from math module would **complain**:

Error: `np.array([1,2,3,4,5,6])` is an **array** but `sin` only accepts a **number**.

The way Numpy helps Python resolves this issue is the introduction of the generalisation of the "array mathematical function" called the ***universal function***.

A ***universal function*** ("***ufunc***"s for short) f is a function that performs elementwise operation on all elements of an array $A = [e_{ij\cdots k}]_{i=1,\cdots,n_1;j=1,\cdots,n_2;\cdots;k=1,\cdots,n_d}$. It has the mathematical formulation:

$$[e_{ij\cdots k}] \xrightarrow{f} [f(e_{ij\cdots k})].$$

"Fast" ufuncs are implemented in **C language** or **Cython**. These are very advanced programming techniques which we will not explore and one can consult Haenel et al. (2015, Section 2.2.2). Instead, we will only explore the friendly way of defining ufuncs using the basic method `np.frompyfunc(func, nin, nout, *[, identity])` (Oliphant 2006). Note that `nin` and `nout` are the number of inputs and number of outputs of the function `func` respectively. The `np.vectorize` can also be use for simple functions with 1 input and 1 output.

The Numpy module provides the extension of Python's `math` module functions to ufunc which works on "array" as defined above.

Mathematical Function	Python/Numpy	Inverse/Pair Function	Python/Numpy
Exponential function, e^x	`np.exp`	Natural logarithmic function, $\ln x$	`np.log`
10^x	`np.power(10,x)`	Logarithm with base 10, $\log_{10} x$	`np.log10`
Hyperbolic sine, $\sinh(x) = \frac{e^x - e^{-x}}{2}$	`np.sinh`	Inverse hyperbolic sine	`np.arcsinh`
Hyperbolic cosine function, $\cosh(x) = \frac{e^x + e^{-x}}{2}$	`np.cosh`	Inverse hyperbolic cosine	`np.arccosh`
Hyperbolic tangent function, $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	`np.cosh`	Inverse hyperbolic tangent	`np.arctanh`
Sine[1]	`np.sin`	Inverse sine function	`np.arcsin`
Cosine[1]	`np.cos`	Inverse cosine function	`np.arccos`
Tangent[1]	`np.tan`	Inverse tangent function	`np.arctan`
Floor	`np.floor`	Ceiling	`np.ceil`
Rounding	`np.round`	-	-

Remark 2.2.11. According to `https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.around.html`, for values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, −0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in the IEEE floating point standard.

[1] angle(s) in **radian**.

Example 2.2.12. Use the array mathematical functions to generate the following arrays in Python:

(a) `A1`=[sin 10°, sin 30°, sin 50°, sin 70°].

(b) `A2`=[tan 6°, tan 42°, tan 66°, tan 78°].

Sample Answer

```
A1 = np.sin(np.radians(np.array([10,30,50,70])))
A2 = np.tan(np.radians(np.array([6,42,66,78])))
```

Example 2.2.13 (Ufunc is used in plotting functions over given points)**.** Use two plotting functions from Topic 4, `plt.plot` and `plt.show` to draw (a) Sine function; (b) Cosine function; and (c) floor function for the domain $[-2\pi, 2\pi]$.

Solution: The question does not say how small is the step size, we will just split $[-2\pi, 2\pi]$ to 100 equal intervals.

```
1 import numpy as np, matplotlib.pylab as plt
2 xr = np.linspace(-2*np.pi,2*np.pi,101)  # x range
3 y1 = np.sin(xr)
4 y2 = np.cos(xr)
5 y3 = np.floor(xr)
6 plt.plot(xr, y1, xr, y2, xr, y3)
7 plt.show()  # This is not needed in Spyder/Jupyter
```

If we have a computer to plot the graph, we can see that floor looks ugly, this is because `plt.plot` just join points and we need break the domain into more intervals to make the plot of floor function nice.

Example 2.2.14. Given a function $\text{sinhc} : \mathbb{R}^+ \to \mathbb{R}^+, \quad x \mapsto \dfrac{\sinh x}{x}$.

(a) Define a Python function implementing the sinhc function.

Solution:

```
def sinhc(x):
    from math import sinh
    return sinh(x)/x
```

(b) Calculate `sinhc(1)`. What is the output? Why?

(c) Calculate `sinhc([1,2,3,4,5,6])`. What is the output? Why?

(d) Define a ufunc `usinhc` to calculate `usinhc([1,2,3,4,5,6])`.

Example 2.2.15 (Final Exam Oct 2018, Q1(b) — **A function and a Ufunc are different**)**.** The dot product of two vectors $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$, $\boldsymbol{x} \cdot \boldsymbol{y}$, is defined as

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

The angle θ between two arrays is defined by the following relation

$$\cos\theta = \frac{\boldsymbol{x}\cdot\boldsymbol{y}}{\sqrt{\boldsymbol{x}\cdot\boldsymbol{x}}\times\sqrt{\boldsymbol{y}\cdot\boldsymbol{y}}}.$$

Implement a **Python function** `theta` to calculate the angle θ (in **degree**) if you are given two arrays `a=[a1,a2,a3,a4]` and `b=[b1,b2,b3,b4]`. You **must** write down the proper import statements. If you use the Numpy module, you must prefix the Numpy functions with "`np.`" or marks will be heavily deducted. Use scientific calculator to find the return value of the Python command `theta([1,2,3,4],[2,1,3,4])` to 4 decimal places. (9 marks)

Sample Programming Solution

```
# Appropriate import when writing scripts                1   mark
import numpy as np
from math import degrees, acos, sqrt
# Able to define a function and return a value:          1.5 marks
# Correct translation of mathematical formula to Python: 2.5 marks
def theta(x, y):
    # We are using the single-line for loop from Topic 1
    # The size of x and y may be different, so we need to
    # check it in real-world programming but it is fine
    # to assume x and y the same size in exam
    num = sum(x[i]*y[i] for i in range(x.size))
    sxx = sqrt(sum(x[i]*x[i] for i in range(x.size)))
    syy = sqrt(sum(y[i]*y[i] for i in range(y.size)))
    return degrees(acos(num/sxx/syy))

print(theta(np.array([1,2,3,4]),np.array([2,1,3,4])))
```

The question also test the understanding of manual calculation and the use of calculator:

$$\theta = \cos^{-1}\frac{2+2+9+16}{\sqrt{1^2+2^2+3^2+4^2}\times\sqrt{2^2+1^2+3^2+4^2}} \quad \text{[1 mark]}$$

$$= \cos^{-1}\frac{29}{30} = 0.258922\times\frac{180^\circ}{\pi} \quad \text{[2 marks]}$$

$$= 14.8351^\circ \quad \text{[1 mark]}$$

§2.3 Arithmetic, Logical and Relational Operations

The basic arithmetic and logical operations for "numbers" (Topic 1) such as `+`, `-`, $\times$, $\div$, power, not, and, or, equality, etc. can be generalised to operate on "arrays". In this section, we will see how to generalise them to "element"-wise operations.

§2.3.1 Arithmetic Operations and Reduction Operations

The basic arithmetic `+`, `-`, $\times$, $\div$ and power for ***two arrays of the same shape*** `A` and `B` are just element-wise addition, subtraction, multiplication, division and power of numbers in the array as:

- `-A`: elementwise negation, $-a_{ij\cdots k}$
- `A + B`: elementwise addition, $a_{ij\cdots k}+b_{ij\cdots k}$
- `A - B`: elementwise subtraction, $a_{ij\cdots k}-b_{ij\cdots k}$
- `A * B`: elementwise multiplication, $a_{ij\cdots k}\times b_{ij\cdots k}$
- `A / B`: elementwise division, $a_{ij\cdots k}/b_{ij\cdots k}$ (can lead to division by zero problem — infinity will be generated)

- `A ** B`: elementwise 'power', $\exp(b_{ij\cdots k} \ln a_{ij\cdots k})$

Together with the ufunc, the array arithmetic allows us to handle computations like $\sin(x^2 + x + 2)$.

In Numpy, the arithmetic operations also work on ***two arrays of the compatible shape***. For example,

- `np.array([1.2, 1.3, 1.4]) + 1`: 1-D array and a number
- Shape (3,4) and shape (4,) are compatible but not shape (3,)
  - `A = np.arange(1,13).reshape(3,4)`
  - `B = np.array([5,4,8])` (Shape = (3,))
  - `C = np.array([9,4,8,7])` (Shape = (4,))
  - `A + C` is OK but `A + B` is **not OK**
  - What is happenning? `C` is expanded to

$$A + C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 9 & 4 & 8 & 7 \end{bmatrix} = \begin{bmatrix} 1+9 & 2+4 & 3+8 & 4+7 \\ 5+9 & 6+4 & 7+8 & 8+7 \\ 9+9 & 10+4 & 11+8 & 12+7 \end{bmatrix}$$

Exercise 1. Write a Python script to plot the functions $y_1 = x^3/2 + 3x^2 - 1$, $y_2 = 2\sin x$ and $y_3 = \sin(2x)$ in one diagram for $-\pi \leq x \leq x$. [Hint: Try to adapt from Example 2.2.13]

Example 2.3.1. Write down two Python commands which allows us to transform the left matrix below to the right matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & -8 & -16 & -24 \end{bmatrix}$$

§2.3.2 Logical Operations

The logical operations for Boolean arrays are similar to the arithmetic operations for numeric arrays. They are just the generalisation of logical operations from Boolean values (True, False) to Boolean arrays (arrays of True and/or False) of compatible arrays)

Let `C` and `D` be Boolean array of the same shape, the element-wise negation, conjunction and disjunction for the Boolean array are:

- Check and make sure that `C.dtype` and `D.dtype` are `bool`.

- `~ C`: elementwise negation, $\sim a_{ij\cdots k}$
- `C & D`: elementwise conjunction, $a_{ij\cdots k} \wedge a_{ij\cdots k}$
- `C | D`: elementwise disjunction, $a_{ij\cdots k} \vee a_{ij\cdots k}$

In Numpy, the logical operations also work on ***two arrays of the compatible shape***. For example, Shape (2,3) and shape (3,) are compatible but not shape (2,)

- `C = np.array([[True, False, True], [False, True, False]])` (Shape = 2x3)
- `D1 = np.array([True, False])` (Shape = 2)
- `D2 = np.array([False, False, True])` (Shape = 3)
- `C & D2` is OK but `C & D1` is **not OK**

Note that Python allows us to use $-$, $*$ and $+$ to denote `~`, `&` and `|` respectively. However, it is not recommended to prevent confusion because Boolean will be converted to integers when other arithmetic operations are involved.

Example 2.3.2 (Final Exam Sept 2015, Q2(a)(iv))**.** What will display if the following commands are executed by Python?

```
np.array([not False, False, not True]) & np.array([True]*3)
```
(2 marks)

Solution: `= [True False False] & [True True True] = [True False False]`

Example 2.3.3. Given the assignments of the variables `p` and `q`:

```
>>> p = np.array([True,True,False,False])
>>> q = np.array([True,False,True,False])
```

Fill in the output / return value for each of the following command.

Command	Output / Return Value
`-p`	
`~p`	
`p&q`	
`p*q`	
`p|q`	
`p+q`	

§2.3.3 Relational Operations

"Booleans" arise when we are "comparing" numbers. Let `A` and `B` be arrays of the same shape. The "comparison" of numbers are accomplished by the relational operations ==, $\sim=$, <, <=, > and >= defined below.

Operation	Definition	Python	R	Octave/ MATLAB
Equal	$a_{ij\cdots k} = a_{ij\cdots k}$	`A == B`	`A == B`	`A == B`
Non-Equal	$a_{ij\cdots k} \neq a_{ij\cdots k}$	`A != B`	`A != B`	`A ~= B`
Greater than	$a_{ij\cdots k} > a_{ij\cdots k}$	`A > B`	`A > B`	`A > B`
Greater equal	$a_{ij\cdots k} \geq a_{ij\cdots k}$	`A >= B`	`A >= B`	`A >= B`
Less than	$a_{ij\cdots k} < a_{ij\cdots k}$	`A < B`	`A < B`	`A < B`
Less equal	$a_{ij\cdots k} \leq a_{ij\cdots k}$	`A <= B`	`A <= B`	`A <= B`

Example 2.3.4 (Final Exam Sept 2013, Q1(c))**.** Given that `x = np.array([1,3,4,2,5,0,-3])` and `y = np.array([6,3,2,4,1,0,6])`, list the results of the following commands (i) to (iii):

(i) `x - 2*(y>3)` (2 marks)

Solution: Let `T` denote True and `F` denote False. The calculation is as follows.

= `x-2*[T F F T F F T] = x-[2 0 0 2 0 0 2] = [-1 3 4 0 5 0 -5]`

(ii) `(x!=0) & (y==0)` (2 marks)

Solution: Let `T` denote True and `F` denote False. The calculation is as follows.

`[T T T T T F T] & [F F F F F T F] = [F F F F F F F]`

(iii) `(x==y) | (y<x)` (2 marks)

§2.3.4 Fancy Indexing with Boolean Array

The "Boolean" array for an array A generated with the use of relational operations can be used as a kind of ***fancy indexing*** called ***Boolean indexing*** for A (refer to `numpy/core/src/multiarray/mapping.c`).

This kind of indexing is widely used in statistics, image processing, signal processing, etc. because it allows us to **filter** out unwanted data in an array.

Example 2.3.5. Consider the 2-D array

$$A = \begin{bmatrix} -1 & 2 & 1 & -3 \\ 2 & -4 & -4 & 0 \\ 0 & 0 & -1 & -2 \end{bmatrix}$$

Write the Python commands to

1. list all the values in A which are **non-negative**.
2. replace the negative values in A by `-10`.

Sample Solution

```
# (a)
A[A>=0]     # or A[~(A<0)]

# Here's how it works for selection:
[-1,  2,  1, -3]  [F, T, T, F]       [  2,  1,
[ 2, -4, -4,  0]  [T, F, F, T] -->     2,      0,
[ 0,  0, -1, -2]  [T, T, F, F]         0, 0]


# (b)
A[A<0] = -10

# Here's how it works for assignment:
[-1,  2,  1, -3]  [T, F, F, T]       [-10,   2,   1, -10]
[ 2, -4, -4,  0]  [F, T, T, F] -->   [  2, -10, -10,   0]
[ 0,  0, -1, -2]  [F, F, T, T]       [  0,   0, -10, -10]
```

Example 2.3.6. The test 2 results of UECM3033 Numerical Methods for the Jan 2018 semester are

11.5, 15.1, 10.8, 14.1, 5.8, 4.1, 15.7, 13.3, 14.6, 5.2, 13.1, 8.6, 8.8, 16.3, 11.7, 13.9, 13.6, 14.5, 11, 14, 13.7, 16.1, 12.1, 9.7, 14.9, 12, 10.5, 12.8, 15.3, 4.8, 13.1, 0, 12.5, 8.8, 14.4, 12.7, 8.8, 11.9, 13.1, 14.4, 7.3, 17.1, 9.3, 11, 13.5, 9, 7.9, 4.7, 13.8, 15.5, 13.2, 8.8, 10.1, 13.3, 9.5, 10.5, 12.6, 14.6, 12.8, 11, 2.7, 6.2, 10.6, 14.5, 13.4, 10.5, 11.3, 14.8, 9.9, 8.8, 14.2, 9.7, 9.4, 9, 13.5, 10.3

The full mark for test 2 is 20 marks and the passing mark is 10. Find all those marks which is below 10. How many students fail?

Sample Solution

```
A = np.array([11.5, ... (just copy the list here) ...,10.3])
failures = A[A<10]
print(f"There are {failures.size} failures out of {A.size}")
```

Remark 2.3.7. One can identify the indices of those that satisfies a predicate `P(x)` with `np.nonzero(P(anarray))`. In Example 2.3.6, `P(x)=(x<10)`, the indices of those who fail the test can be found by `np.nonzero(A<10)`.

Example 2.3.8 (Final Exam Sept 2013, Q1(b) with modification)**.** Write a Python script to perform the following actions:

- Generate a 2-by-3 array of random numbers using `rand` command and,
- Move through the array, element by element, and set any value that is less than 0.2 to 0 and any value that is greater than or equal to 0.2 to 1. (9 marks)

Solution: The intention of the lecturer who set the question is to ask you to express this in for loop but in reality, everyone use the array functions from Topic 2 to achieve the stated requirements.

Open up a notepad (or Spyder), type in the following text and then save it:

```
1 import numpy as np
2 A = np.random.rand(2,3) # A 2x3 array of random numbers
3 A[A<0.2]  = 0
4 A[A>=0.2] = 1
```

Run the above script a few times and explain what do you observe?

Example 2.3.9 (From `https://www.python-course.eu/numpy_masking.php`)**.** Extract from the array `B=np.array([3,4,6,10,24,89,45,43,46,99,100])` those numbers

- which are not divisible by 3;
- which are divisible by 5;
- which are divisible by 3 and 5;
- which are divisible by 3 and set them to 42.

Example 2.3.10 (A more complicated example — Fractal, something popular in 1990s?)**.** Use 2D array to draw the `https://en.wikipedia.org/wiki/Mandelbrot_set` for the domain $[-2.5, 1.5] \times [-1.5, 1.5]$.

Sample Solution

```
import numpy as np, matplotlib.pylab as plt
N = 200
xa, xb = -2.5, 1.5
x = np.linspace(xa, xb, round(abs(xb-xa))*N+1)
ya, yb = -1.5, 1.5
y = np.linspace(ya, yb, round(abs(yb-ya))*N+1)
XX, YY = np.meshgrid(x, y)
#
# The mesh grid generates (XX, YY) which represents
# y axis
# |                      x axis --->
# V
#  (-2.5, -1.5  ) (-2.495, -1.5  ) ... (1.495, -1.495) (1.5, -1.5  )
#  (-2.5, -1.495) (-2.495, -1.495) ... (1.495, -1.495) (1.5, -1.495)
#  ...
#  (-2.5,  1.5  ) (-2.495,  1.5  ) ... (1.495,  1.5  ) (1.5,  1.5  )
#
CC = XX + YY*1j   # Create complex numbers matrix
ZZ = np.zeros_like(CC)
maxit = 20   # Try 20 iterations in for loop
R     = 2    # Radius for divergence
divtime = maxit + np.zeros(ZZ.shape, dtype=int)
for i in range(maxit):
    ZZ = ZZ**2 + CC
    diverge = abs(ZZ) > R                    # who is diverging
    div_now = diverge & (divtime == maxit)  # who is diverging now
    divtime[div_now] = i                     # Boolean indexing
    ZZ[diverge] = R                          # avoid diverging too much

# https://matplotlib.org/stable/tutorials/colors/colormaps.html
# Change colour map is 'viridis' to 'Spectral'
plt.imshow(divtime, cmap='Spectral')
plt.show()
```

§2.4 Array Reduction Operations

When we want to work on the elements of arrays along some **axis** or multiple axes, we can regard as **reducing** the array data to some values. We will explore some classes of reduction operations below.

- **'Reduction' Operations for Filtering and Construction**:
  - `.real`, `.imag`: takes the real and imaginary parts of every element for (complex) numeric matrices.
  - `np.choose(J, A)` picks `A[J[i]]` into `J`. It complements `np.compress()`, `np.select()`, `np.diag(Arr1D)`, `np.diagonal()`, `np.extract()`, etc.
  - `np.putmask(A, mask, values)`: works similar to Boolean masking. Closely related to `np.take()`, `np.place()`, `np.put()`, `np.copyto()`.

    ```
    np.putmask(A, A<0, 0)    # Same as A[A<0] = 0
    ```

  - Signal processing functions: `np.correlate(x,y)` correlates two 1-D arrays ($z_j = \sum_{i=\max(j-M,0)}^{\min(j,K)} x_i y_{j+i}$, $j = 0, \cdots, K+M$); `np.convolve(x,y)` convolves two 1-D

arrays ($z_j = \sum_{i=\max(j-M,0)}^{\min(j,K)} x_i y_{j-i}$, $j = 0, \cdots, \max(K, M)$) Here x and y are two 1-D arrays with $K = x.\texttt{size} - 1$ and $M = y.\texttt{size} - 1$.

- **Ordering 'Reduction' Operations**: `argmax` (index of largest value), `argmin` (index of smallest value), `argsort` (return indices of sorted array), `min`, `max`, `ptp` (range of values (maximum – minimum) along an axis), `searchsorted`, `sort`.
- **Statistical 'Reduction' Operations**: The functions `cov`, `mean`, `median`, `std`, `var` will be useful for Probability and Statistics I.
  - `np.sum(`X`)`
  - `np.mean(`X`)`: When X is simulated by data $x_1, \cdots, x_n$, then $\mathbb{E}[X] \approx \dfrac{x_1 + \cdots + x_n}{n} = \overline{X}$.
  - `np.median(`X`)`: Find the median of data X.
  - `np.var(`X`)` (and `std(`X`)` $= \sqrt{\texttt{var}(X)}$): By default, it is the population variance (and standard deviation)

    $$\mathrm{Var}[X] = \mathbb{E}[|X - \mathbb{E}[X]|^2] \approx \frac{(x_1 - \overline{X})^2 + \cdots + (x_n - \overline{X})^2}{n}.$$

    Note that for sample variance (and sample population), the n needs to be changed to $n - 1$ (set `ddof=1`).
  - `np.cov(`X`)`: Compute the covariance matrix of data in X based on the mathematical formulation:

    $$Cov[X] = \mathbb{E}[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^H].$$

Example 2.4.1 (General and Statistical 'Reduction' Operations)**.** Consider the array

$$M = \begin{bmatrix} 6 & 9 & 12 & 4 & 3 & 0 \\ 4 & 4 & 15 & 2 & 1 & 1 \\ 2 & 1 & 18 & -5 & 8 & 2 \\ -6 & -4 & 21 & 1 & -5 & 2 \end{bmatrix}$$

Let us investigate the sum, prod, cumsum, cumprod, min, max, range, mean, var (population variance), std (population standard deviation), etc. along the whole array, along the row and along the column.

Solution: Let's investigate how the axis work with the various given reduction operations.

```
M = np.array([[ 6,  9, 12,  4,  3, 0],
              [ 4,  4, 15,  2,  1, 1],
              [ 2,  1, 18, -5,  8, 2],
              [-6, -4, 21,  1, -5, 2]])

#
# Along the whole array
#
M.sum()           # 96 (all numbers add)
M.prod()          # 0  (all numbers multiply)
M.cumsum()        # 6, 6+9, 6+9+12, ...
M.cumprod()       # 6, 6*9, 6*9*12, ...
M.min()           # -6
M.max()           # 21
M.ptp()           # 27
M.mean()          # 4.0
M.var()           # 46.25
M.std()           # 6.800735254367722
```

```
#
# NOTE: There is a 'nan' version for the above commands which
# skips nan, e.g. np.nanmean(x), np.nanvar(x), etc.
#

#
# Along the rows (axis = 1)
#
M.sum(axis=1)  # It will sum along the row return 1-D array
M.prod(axis=1, keepdims=1)   # use keepdims=1 if we want 2-D array
M.cumsum(axis=1)     # [6,6+9,...], [4,4+4,...], [2,2+1,...], ...
M.cumprod(axis=1)    # [6,6*9,...], [4,4*4,...], [2,2*1,...], ...
M.min(axis=1)        # [ 0,  1, -5, -6]
M.max(axis=1)        # [12, 15, 18, 21]
M.ptp(axis=1)
M.mean(axis=1)
M.var(axis=1)        # for sample variance, use ddof=1
M.std(axis=1)

#
# Along the columns (axis = 0)
#
M.sum(axis=0, keepdims=1)
M.prod(axis=0)
M.cumsum(axis=0)
M.cumprod(axis=0)
M.min(axis=0)
M.max(axis=0)
M.ptp(axis=0)
M.mean(axis=0)
M.var(axis=0)
M.std(axis=0)
```

Example 2.4.2 (Final Exam Sept 2015, Q1(b)(i))**.** Write down and explain the values of `C` for the following commands

```
import numpy as np
A = np.array([4,6,8],dtype='double')
B = np.array([2,0,4])
C = np.sum(A/B)
```

(3 marks)

Solution: `C = sum([4/2 6/0 8/4]) = sum([2 Inf 2]) = Inf`

Example 2.4.3. Suppose you have keyed in an array of the following exam data (out of 30 marks):

10 24 NaN 22 25 17 23

The "NaN" indicates that a student is absent. Find the average and standard deviation by filtering "NaN".

Sample Solution

```
d = np.array([10, 24, np.nan, 22, 25, 17, 23])
dbar = np.nanmean(d)
ddev = np.nanstd(d)
ans = "The average marks is {avg:.2f} / 30 and the standard"
ans += " deviation is {s:.2f}"
print(ans.format(avg=dbar, s=ddev))
```

Example 2.4.4 (Final Exam Oct 2018, Q1(a), CO1)**.** The output of the Python commands below

```
>>> import numpy as np
>>> A = np.arange(1,36).astype('float').reshape(5,7)
>>> print(A)
```

is

```
[[  1.   2.   3.   4.   5.   6.   7.]
 [  8.   9.  10.  11.  12.  13.  14.]
 [ 15.  16.  17.  18.  19.  20.  21.]
 [ 22.  23.  24.  25.  26.  27.  28.]
 [ 29.  30.  31.  32.  33.  34.  35.]]
```

Use the above information to write down the output to the following Python commands for item (i) to item (iv).

(i) `print(A[2,:])` (2 marks)

(ii) `print(A[1:4,3:5])` (3 marks)

(iii) `print(A[2:4,4:6].mean())` (5 marks)

(iv) `print(A[:,2]>10)` (3 marks)

(v) Write down the Python command to **count** the number of elements in `A` who are larger than 20. (3 marks)

Example 2.4.5. (a) Write down the Python command to subtract the each column of a matrix `A` by the mean of the data of each column vector.

(b) The 'shortest' Python command to subtract the each row of a matrix `A` by the mean of the data of each row vector is probably

```
(A.T - A.mean(1)).T
```

Do you know other (slightly longer) Python command which achieves the same outcome for the matrix `A`?

Example 2.4.6 (Course Outcome 3: Write program script)**.** The following function is used to generate a moving sequence (with a particular window size, by default 4) for a one-dimensional array `a`.

```
1 def rolling(a, window=4):
2     n = a.size
3     newarray = np.zeros((n-window+1,window))
4     for i in range(n-window+1):
5         newarray[i,:] = a[i:i+window]
6     return newarray
```

When the one-dimensional array is $\boldsymbol{x} = [x_1, x_2, x_3, x_4, \cdots, x_n]$, the return moving sequence of `rolling(`$\boldsymbol{x}$`)` with a widow size 4 is

$$[[x_1, x_2, x_3, x_4],\ [x_2, x_3, x_4, x_5],\ \cdots, [x_{n-3}, x_{n-2}, x_{n-1}, x_n]].$$

(i) If `a = np.array([1.1, 1.34, 1.17, 1.06, 1.06, 0.94])`, write down the output of Python command `rolling(a)`.

Solution

```
array([[1.1 , 1.34, 1.17, 1.06],
       [1.34, 1.17, 1.06, 1.06],
       [1.17, 1.06, 1.06, 0.94]])
```

(ii) Define a Python function `moving_average` to calculate moving average of $\boldsymbol{x}$ with a window of 4 which returns the following array:

$$[\frac{x_1 + x_2 + x_3 + x_4}{4},\ \frac{x_2 + x_3 + x_4 + x_5}{4},\ \cdots,\ \frac{x_{n-3} + x_{n-2} + x_{n-1} + x_n}{4}]$$

based on the moving sequence `rolling(`$\boldsymbol{x}$`)`. Write down the output of the Python command `print(moving_average(a))` where `a` is given in part (i).

(iii) Explain how to calculate moving variance of `a` in part (i) with a window of 4.

Example 2.4.7 (Final Exam Oct 2018, Q2(b), CO3)**.** The following function from a program script is used to generate a moving sequence for a one-dimensional array

```
1 def rolling(a, window=4):
2     n = a.size
3     newarray = np.zeros((n-window+1,window))
4     for i in range(n-window+1):
5         newarray[i,:] = a[i:i+window]
6     return newarray
```

When the one-dimensional array is $\boldsymbol{x} = [x_1, x_2, x_3, x_4, \cdots, x_n]$, the return moving sequence of `rolling(`$\boldsymbol{x}$`)` is

$$[[x_1, x_2, x_3, x_4],\ [x_2, x_3, x_4, x_5],\ \cdots, [x_{n-3}, x_{n-2}, x_{n-1}, x_n]].$$

(i) If `a = np.array([0.95, 0.87, 0.87, 0.98, 1.04, 1.08])`, write down the output of `rolling(a)`.

(3 marks)

(ii) Define a Python function `moving_average` to calculate moving average of $\boldsymbol{x}$ with a window of 4 which returns the following array:

$$[\frac{x_1 + x_2 + x_3 + x_4}{4},\ \frac{x_2 + x_3 + x_4 + x_5}{4},\ \cdots, \frac{x_{n-3} + x_{n-2} + x_{n-1} + x_n}{4}]$$

based on the moving sequence `rolling(`$\boldsymbol{x}$`)`. Write down the output of the Python command `print(moving_average(a))` where `a` is given in part (i). (5 marks)

(iii) Explain how to calculate moving variance of `a` in part (i) with a window of 4. (2 marks)

Remark 2.4.8. The series in Python's pandas module since 0.20 supported a limited rolling function.

Example 2.4.9 (Final Exam Sept 2019, Q1)**.** (a) Given

$$\mathtt{A} = \begin{bmatrix} 4 & 9 & 8 & 0 & 2 & 9 \\ 3 & 1 & 9 & 1 & 2 & 8 \\ 7 & 2 & 2 & 3 & 7 & 8 \\ 3 & 5 & 0 & 7 & 9 & 7 \\ 9 & 4 & 6 & 7 & 9 & 8 \end{bmatrix}.$$

Use the above information to **execute** the following Python commands for item (i) to item (iv) and write down the output of the execution.

(a) `print(A[:,1])` (2 marks)

> **Solution**: `[9 1 2 5 4]` ..... [2 marks]

(b) `print(A[1:4,[1,2,3]])` (3 marks)

> **Solution**:
> ```
> [[1 9 1]
>  [2 2 3]
>  [5 0 7]]
> ```
> ..... [3 marks]

(c) `print(A[A<5].sum())` (4 marks)

> **Solution**: $(4 + 0 + 2) + (3 + 1 + 1 + 2) + (2 + 2 + 3) + (3 + 0) + 4 = 27$ [4 marks]

(d) `print(A[:4,:3].sum(axis=0))` (3 marks)

> **Solution**: $[4 + 3 + 7 + 3, 9 + 1 + 2 + 5, 8 + 9 + 2 + 0] = [17, 17, 19]$ [3 marks]

(e) Write down the Python command which gives the mean of rows in `A` after **execution**. (2 marks)

> **Solution**: `print(A.mean(axis=1))` ..... [2 marks]

(f) Write down the warning message that the command

```
print(A[1,:]/A[0,:].astype(np.float64))
```

will raise when it is executed. (2 marks)

> **Solution**: Since `A[0,3]` is zero, a division by zero error will be produced.
> ..... [2 marks]

(b) Use Numpy array operations such as `np.arange`, etc. to write a computer program in no more than 3 lines and without using any semicolon to print the following output:

```
 1     1     1     1
 2     4     8    16
 3     9    27    81
 4    16    64   256
 5    25   125   625
 6    36   216  1296
 7    49   343  2401
 8    64   512  4096
 9    81   729  6561
10   100  1000 10000
```

(4 marks)

Sample Solution

```
col1 = np.arange(1,11).reshape(10,1)              # [1.5 marks]
B = np.hstack((col1, col1**2, col1**3, col1**4))  # [2   marks]
print(B)                                          # [0.5 mark ]
```

§2.5 Linear Algebra Operations

A 1-D array can be regarded as a ***vector***.

What makes "vectors" special in mathematics (not computer program) are the linear algebra operations associated with them:

- **length**: `linalg.norm(x[, ord, axis, keepdims])`
- **dot-product**: `np.vdot(v1, v2)` (the **angle** is the arc-cosine of the dot product).

A 2-D array can be regarded as a ***matrix*** (there is a "matrix" type in Numpy but is not recommended).

What makes "matrices" special in mathematics (not computer program) are the linear algebra operations associated with them. The following are operations which work for two n-D arrays of compatible shapes (including 1-D arrays).

- `np.linalg.matrix_power(A, n)`: Raise a square matrix to the (integer) power `n`.
- `np.matmul(A, B[, out])` or `A@B`: Matrix product of two arrays.
- `np.dot(A, B[, out])`: Dot product $z[I,J,j] = \sum_k A[I,k]B[J,k,j]$ of `A` and `B`.
- `np.linalg.multi_dot(arrays)`: Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
- `np.inner(a, b)`: Inner product of two arrays, i.e. $\sum_{i=0}^{K} x_i y_i$ if $K = M$.
- `np.outer(a, b[, out])`: Compute the outer product of two vectors, i.e. $[x_i y_j]$.
- `np.tensordot(a, b[, axes])`: Compute tensor dot product along specified axes for arrays $\geq$ 1-D.
- `np.einsum(subscripts, *operands[, out, dtype, ...])`: Evaluates the Einstein summation convention on the operands.
- `np.einsum_path(subscripts, *operands[, optimize])`: Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.
- `np.kron(A,B)`: Kronecker product of two arrays, giving $[a_{ij\ldots k}B]$.

Note that many of the linear algebra routines in `np.linalg` are able to compute results for several matrices at once, if they are **stacked** into the same array. This is indicated in the documentation via input parameter specifications such as `A : (..., M, M) array_like`. This means that if for instance given an input array `A.shape == (N, M, M)`, it is interpreted as a "stack" of `N` matrices, each of size `M`-by-`M`. Similar specification applies to return values, for instance the determinant has `det : (...)` and will in this case return an array of shape `det(a).shape == (N,)`. This generalises to linear algebra operations on higher-dimensional arrays: the last 1 or 2 dimensions of a multidimensional array are interpreted as vectors or matrices, as appropriate for each operation.

Example 2.5.1 (Linear Algebra Operations on 1-D arrays)**.** Study the following linear algebra operations:

```
x = np.array([3,-1,2,-4])
y = np.array([5,9,7])
z = np.array([5,7,3,1])
#
# Length of x, denoted by |x|
# = sqrt(3^2 + (-1)^2 + 2^2 + (-4)^2) = 5.4772
#
np.linalg.norm(x)
```

```
#
# Angle between vectors of the same size, x and z
# = arccos( (3*5 + (-1)*7 + 2*3 + (-4)*1) / |x| / |z| )
#
np.arccos(np.vdot(x,z)/np.linalg.norm(x)/np.linalg.norm(z))

#
# The np.outer is a linear algebra coming from
# the multiplication table.
# [ 3]                [15, 27, 21]
# [-1]                [ 5,  9,  7]
# [ 2] o [5, 9, 7] = [10, 18, 14]
# [-4]                [20, 36, 28]
#
np.outer(x,y)
```

Example 2.5.2 (Final Exam Oct 2018, Q1(b))**.** The dot product of two vectors $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$, $\boldsymbol{x} \cdot \boldsymbol{y}$, is defined as

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

The angle θ between two arrays is defined by the following relation

$$\cos\theta = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\sqrt{\boldsymbol{x} \cdot \boldsymbol{x}} \times \sqrt{\boldsymbol{y} \cdot \boldsymbol{y}}}.$$

Implement a **Python function** `theta` to calculate the angle θ (in **degree**) if you are given two arrays `a=[a1,a2,a3,a4]` and `b=[b1,b2,b3,b4]`. You **must** write down the proper import statements. If you use the Numpy module, you must prefix the Numpy functions with "`np.`" or marks will be heavily deducted. Use scientific calculator to find the return value of the Python command `theta([1,2,3,4],[2,1,3,4])` to 4 decimal places. (9 marks)

Try to work out the answer using dot product and vector norms.

Example 2.5.3 (Linear Algebra Operations on 2-D arrays)**.** Consider

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 7 & 8 \\ 8 & 7 \end{bmatrix} \quad C = \begin{bmatrix} 3 & -2 \end{bmatrix}.$$

Study the following linear algebra operations:

```
A = np.array([[1, 2],[3, 4],[5, 6]])
B = np.array([[7, 8],[8, 7]])
C = np.array([3,-2])
#
# B^4 = B x B x B x B = [25313, 25312]
#                       [25312, 25313]
B @ B @ B @ B
np.linalg.matrix_power(B, 4)
```

```
#          [23, 22]
# A x B = [53, 52]
#          [83, 82]
A @ B    # Other: np.matmul(A, B), np.dot(A, B), np.inner(A,B)

#
# Python will cleverly regard C as column matrix
# when B (2x2 matrix) is multiplied to C
#
B @ C    # [ 5, 10]

#                                          [ 7,  8, 14, 16]
#                               [ 1B, 2B ]   [ 8,  7, 16, 14]
# Kronecker product A (x) B = [ 3B, 4B ] = [21, 24, 28, 32]
#                               [ 5B, 6B ]   [24, 21, 32, 28]
#                                          [35, 40, 42, 48]
#                                          [40, 35, 48, 42]
np.kron(A, B)
```

Example 2.5.4. Explain which of the Python/Numpy instruction is most appropriate achieved the following results.

(a) Generate the results of multiplication tables for 2 to 9.

(b) Let $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. Use A to generate a matrix like this $\begin{bmatrix} 6 & 8 & 10 & 3 & 4 & 5 \\ 10 & 8 & 6 & 5 & 4 & 3 \\ 3 & 4 & 5 & 6 & 8 & 10 \\ 5 & 4 & 3 & 10 & 8 & 6 \end{bmatrix}$.

Example 2.5.5 (Final Exam Sept 2019, Q2)**.**

Given the matrix

$$\mathtt{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

(a) **Execute** the following Python commands for item (i) to item (v) and write down the output of the execution.

(a) `print(M * M)` (3 marks)

Solution: $\begin{bmatrix} 1 & 4 \\ 9 & 16 \\ 25 & 36 \end{bmatrix}$ .....[3 marks]

(b) `print(M @ M.T)` (3 marks)

Solution: $\begin{bmatrix} 5 & 11 & 17 \\ 11 & 25 & 39 \\ 17 & 39 & 61 \end{bmatrix}$ .....[3 marks]

(c) `print(M[[2,1,0,1,2],:][:,[1,0,0,1]])` (4 marks)

Solution: $\begin{bmatrix}5 & 6\\3 & 4\\1 & 2\\3 & 4\\5 & 6\end{bmatrix} \rightarrow \begin{bmatrix}6 & 5 & 5 & 6\\4 & 3 & 3 & 4\\2 & 1 & 1 & 2\\4 & 3 & 3 & 4\\6 & 5 & 5 & 6\end{bmatrix}$ ..... [4 marks]

(d) `print(M[:2,:]==M[[2,1],:])` (2 marks)

Solution: $\begin{bmatrix}False & False\\True & True\end{bmatrix}$ ..... [2 marks]

(e) `print((M<3)|(M>4))` (3 marks)

Solution: $\begin{bmatrix}True & True\\False & False\\True & True\end{bmatrix}$ ..... [3 marks]

(b) Write a Python program with no more than 3 lines to produce the following matrices from `M`:

$$M_1 = \begin{bmatrix}-2.5 & -1.5\\-0.5 & 0.5\\1.5 & 2.5\end{bmatrix}, \quad M_2 = \begin{bmatrix}-2 & -2\\0 & 0\\2 & 2\end{bmatrix}, \quad M_3 = \begin{bmatrix}-0.5 & 0.5\\-0.5 & 0.5\\-0.5 & 0.5\end{bmatrix}$$

by using the Numpy vector operation in the Python computer software. Note that M_1 is `M` subtracted by the mean of all values in `M`, M_2 is a matrix such that each column in `M` being subtracted by the mean of corresponding column, M_3 is a matrix such that each row in `M` being subtracted by the mean of corresponding row. Note that your program must work when `M` is changed to an arbitrary $m \times n$ matrix. (5 marks)

Sample Solution

```
M1 = M - M.mean()                                  # [1 mark ]
M2 = M - M.mean(axis=0))                           # [2 marks]
M3 = M - M.mean(axis=1)[:,None])                   # [2 marks]
```

§2.6 Matrix Equation Solvers

In science and engineering, we often encounter the equations involving matrices (2-D array), which can be called ***matrix equations***.

A famous matrix equation is the linear algebra problem:

$$AX = B \tag{2.1}$$

where A is an $m \times n$ matrix, X is an $n \times k$ matrix and B is an $m \times k$ matrix. X is unknown whereas A and B need to be given.

The methods for solving many matrix equations are available in Python through:

```
from scipy import linalg
```

Mathematicians have solved (2.1) with $\boxed{m = n \ll 10^4}$ using the Gaussian elimination method (and Cholesky method when the matrix A is positive definite) leading to the following functions in Python:

- `linalg.solve(A, B)`: Solve (2.1).
- `linalg.lu(A)`: Find the PLU decomposition of `A`.
- `linalg.lu_factor`: It should be used followed by repeated applications of the com-

mand `linalg.lu_solve` to solve the (2.1). Similarly, the `linalg.cho_factor` and `linalg.cho_solve` pair are used for positive matrices.

- `linalg.inv(A)`: Compute the (multiplicative) **inverse** of a matrix A. It is the same as solving (2.1) with B being the identity matrix.
- `linalg.cholesky(a)`: Cholesky decomposition.

Example 2.6.1 (Final Exam Sept 2014, Q4(a))**.** Given the linear system

$$\begin{aligned} 3x_1 + 7x_2 - 2x_3 + 3x_4 - x_5 &= 37 \\ 4x_1 + 3x_5 &= 40 \\ 5x_3 - 4x_4 + x_5 &= 12 \\ 2x_1 + 9x_3 + 4x_4 + 3x_5 &= 14 \\ 5x_4 + 8x_5 &= 20 \end{aligned}$$

Write a Python script to solve the linear system.

Sample Python Script

```
import numpy as np
from scipy import linalg
A = np.array([[3, 7, -2,  3, -1],
              [4, 0,  0,  0,  3],
              [0, 0,  5, -4,  1],
              [2, 0,  9,  4,  3],
              [0, 0,  0,  5,  8]])
x = linalg.solve(A, [37, 40, 12, 14, 20])
```

Example 2.6.2 (Final Exam Sept 2020 during MCO, Q1)**.** (a) Given that `A` stores the following matrix

$$\begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 15 & 8 & 1 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 6 & 24 & 6 & 1 & 0 \\ 0 & 0 & 6 & 0 & 0 & 1 & 8 & 15 & 4 & 4 \\ 1 & 0 & 0 & 3 & 0 & 0 & 8 & 4 & 18 & 5 \\ 2 & 3 & 0 & 0 & 5 & 0 & 0 & 8 & 6 & 24 \\ 29 & 3 & 1 & 0 & 0 & 5 & 0 & 0 & 3 & 7 \\ 3 & 17 & 5 & 6 & 0 & 0 & 7 & 0 & 0 & 2 \\ 4 & 4 & 17 & 3 & 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 8 & 2 & 25 & 4 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 7 & 1 & 16 & 0 & 0 & 0 & 0 & 7 \end{bmatrix}.$$

(a) Write down the output of the Python command `A[:,[3,5,2,4]]`. Determine if it is the same as `A[[3,5,2,4]]` and explain the difference. (1 mark)

Solution:

```
[[ 0 15  0  0]
 [ 0  6  0  0]
 [ 0  1  6  0]
 [ 3  0  0  0]
 [ 0  0  0  5]
 [ 0  5  1  0]
 [ 6  0  5  0]
 [ 3  0 17  4]
 [25  0  2  4]
 [ 1  0  7 16]]
```

.....[0.8 mark]

`A[:,[3,5,2,4]]` and `A[[3,5,2,4]]` are different because the former picks the

columns while the later pick the rows. .....[0.2 mark]

(b) Write the Python command to pick all the odd rows and even columns from `A` and write down the output of your command. (1 mark)

Solution: `A[::2,1::2]` .....[0.7 mark]

```
[[ 0  0  0 24  1]
 [ 1  0  0  8 18]
 [29  1  0  0  3]
 [ 4 17  4  0  0]
 [ 0  7 16  0  0]]
```

.....[0.3 mark]

(c) Write the Python command to pick the intersection of the second, fifth, third columns and of the eighth, fifth and seventh rows in the given order and write down the output of your command. (1 mark)

Solution: `A[:,[1,4,2]][[7,4,6],:]` .....[0.7 mark]

```
[[ 4  4 17]
 [ 3  5  0]
 [17  0  5]]
```

.....[0.3 mark]

(d) Write the Python command to arrange the given matrix `A` into the following diagonally dominant form:

$$\begin{bmatrix} 15 & 8 & 1 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 6 & 24 & 6 & 1 & 0 & 0 & 6 & 0 & 0 & 0 \\ 1 & 8 & 15 & 4 & 4 & 0 & 0 & 6 & 0 & 0 \\ 0 & 8 & 4 & 18 & 5 & 1 & 0 & 0 & 3 & 0 \\ 0 & 0 & 8 & 6 & 24 & 2 & 3 & 0 & 0 & 5 \\ 5 & 0 & 0 & 3 & 7 & 29 & 3 & 1 & 0 & 0 \\ 0 & 7 & 0 & 0 & 2 & 3 & 17 & 5 & 6 & 0 \\ 0 & 0 & 6 & 0 & 0 & 4 & 4 & 17 & 3 & 4 \\ 0 & 0 & 0 & 8 & 0 & 0 & 8 & 2 & 25 & 4 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 7 & 1 & 16 \end{bmatrix}$$

(0.5 mark)

Solution: `A[:,[5,6,7,8,9,0,1,2,3,4]]` .....[0.5 mark]

(e) For an $n \times n$ matrix `A`, it is said to be *diagonally dominant* if for each row the absolute value of the diagonal element is larger than the sum of the absolute value of the rest of the elements in the row:

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|, \quad i = 1, 2, \cdots, n.$$

Write a Python function `is_diag_domin(A)` which determines whether the matrix A is diagonally dorminant. The function with return True if the matrix A is diagonally dorminant, False if the matrix A is not diagonally dorminant, and None if the matrix is not square. (1 mark)

Sample Solution

```
1 def is_diag_domin(A):
```

```
    N = A.shape[0]
    for i in range(N):
        S = sum(abs(A[i,j]) for j in range(N) if j != i)
        if abs(A[i,i]) <= S:
            print("i=",i)
            return False
    return True

#import q1
#print(is_diag_domin(q1.AA))
```

(b) Given that three 3×3 matrices $P = \begin{bmatrix} 5 & 8 & 8 \\ 6 & -9 & -8 \\ 6 & -5 & 1 \end{bmatrix}$, $Q = \begin{bmatrix} 2 & 2 & -2 \\ 7 & 8 & -2 \\ 0 & 2 & 2 \end{bmatrix}$ and $R = \begin{bmatrix} -2 & -8 & 8 \\ -8 & -5 & 8 \\ 6 & -9 & 4 \end{bmatrix}$.

(a) Write down the Python command to find the inverse matrix of Q, Q^{-1}. (0.5 mark)

Solution: The Python command to find Q^{-1} is `np.linalg.inv(Q)` or `np.linalg.solve(Q,np.eye(Q.shape[0]))`. The output is ..... [0.5 mark]

$$\begin{bmatrix} -1.25 & 0.5 & -0.75 \\ 0.875 & -0.25 & 0.625 \\ -0.875 & 0.25 & -0.125 \end{bmatrix}$$

(b) Write down the Python command to find matrix L if $P^3LQ = R$. Write down the **matrix** L. (1 mark)

Solution: $L = (P^3)^{-1}RQ^{-1}$ ..... [0.7 mark]

```
L = linalg.inv(P@P@P) @ R @ linalg.inv(Q)
L = linalg.solve(np.linalg.matrix_power(P,3),R)@linalg.inv(Q)
```

The matrix L is

$$\begin{bmatrix} 0.08603119 & -0.04214438 & 0.07957573 \\ 0.21360577 & -0.09753974 & 0.18129806 \\ -0.24895274 & 0.11235113 & -0.20818642 \end{bmatrix}$$ [0.3 mark]

(c) Suppose the 3×3 matrices E, F, G, H satisfies

$$\begin{bmatrix} P & Q \\ Q & R \end{bmatrix}^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

First, find the matrix H by writing down the appropriate Python commands. Then, write down the appropriate Python command(s) to show that

$$(R - QP^{-1}Q)^{-1} = H.$$ (1 mark)

Solution: After `from scipy import linalg`, the command

$$H = \texttt{linalg.inv(R} - \texttt{Q@linalg.inv(P)@Q)}$$

allows us to obtain

$$H = \begin{bmatrix} 0.26819736 & -0.15480007 & -0.07891041 \\ 0.69421553 & -0.3532685 & -0.42828374 \\ 1.00692917 & -0.48176952 & -0.51330189 \end{bmatrix} \quad \text{[1 mark]}$$

[Total : 7 marks]

For some linear system (2.1) with **special square matrices** such as the **Toeplitz matrix**:

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

It can be generated using

```
linalg.toeplitz([a1, a2, a3, ..., an], [b0, b1, b2, ..., bm])
```

Mathematicians have developed algorithms to solve (2.1) much faster leading to the following functions in Python:

- `linalg.solve_toeplitz(c_or_cr, b, check_finite=True)`
- Other special cases are ignored.

Example 2.6.3. Construct a Toeplitz matrix from the 1-D arrays c=[2,3,4,5] and r=[500,6,7,8,9,10].

Answer

```
>>> print(linalg.toeplitz([2,3,4,5],[500,6,7,8,9,10]))

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

Example 2.6.4. Solve the Toeplitz system

$$\begin{bmatrix} 1 & -1 & -2 & -3 \\ 3 & 1 & -1 & -2 \\ 6 & 3 & 1 & -1 \\ 10 & 6 & 3 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 5 \end{bmatrix}$$

We can solve this problem similar to Example 2.6.1. However, when we encounter a large Toeplitz system which uses a lot of memory, we should solve it as follows.

Sample Solution

```
# MUST MAKE SURE linalg.toeplitz(c, r) is the same as
# left matrix in order to use this.
c = np.array([1,3,6,10])    # first column of left matrix
r = np.array([1,-1,-2,-3])  # first row    of left matrix
b = np.array([1,2,2,5])     # right column matrix
x = linalg.solve_toeplitz((c, r), b)
```

Mathematicians have solved the general (2.1) with no restrictions (except that that they cannot be too large because computer memory is limited) on m and n (the price to pay is longer computation time) using the SVD method or QR method leading to the following functions in Python:

- `linalg.lstsq(A, B[, rcond])`: Return the least-squares solution X, the residue $B - AX$, the rank and singular values of A to (2.1).
- `linalg.svd(a[, full_matrices, compute_uv])`: Singular value decomposition (SVD).
- `linalg.qr(a[, mode])`: Compute the qr factorisation of a matrix.
- `linalg.pinv(a[, rcond])`: Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Example 2.6.5. Write down the Python script to solve the following problem:

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 19 & 1 \\ 3 & 2 \end{bmatrix}.$$

Sample Solution

```
from scipy import linalg   # Mentioned earlier
A = np.array([[-2,11],[17,-19]])
B = np.array([[19,1],[3,2]])
X = linalg.solve(A, B)
print("X=",X)
```

The solution is

$$X = \begin{bmatrix} 2.64429530 & 0.27516779 \\ 2.20805369 & 0.14093960 \end{bmatrix}.$$

We can verify this using the linear algebra knowledge from SPM.

Example 2.6.6. Write down the Python script to solve the following problem:

$$\begin{bmatrix} -2 & 11 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 19 & 1 \end{bmatrix}.$$

Sample Solution

```
A = np.array([[-2,11]])
B = np.array([[19,1]])
X,_,Rank,Sing = linalg.lstsq(A, B)
print("X=",X)   # Many solutions but only one return
```

Mathematicians have solved the **(right) matrix eigenvalue problem**:

$$AX = X\Lambda \tag{2.2}$$

making the following solvers available to us:

- `linalg.eig(A)`: Compute the eigenvalues and normalised right eigenvectors of the square array `A`.
- `linalg.eigh(A)`: Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix `A`.

Mathematicians have solved the **(right) generalised matrix eigenvalue problem**:

$$AX = BX\Lambda \tag{2.3}$$

making the following solvers available to us:

- `linalg.eigvals(A,B)`: Compute the eigenvalues of a general matrix.
- `linalg.eigvalsh(A,B)`: Compute the eigenvalues of Hermitian or real symmetric matrices `A` and `B`.

Example 2.6.7. Write down the Python script to find the least square solution the following problem:

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \\ 6 & 6 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 19 & 1 \\ 3 & 2 \\ 6 & 7 \end{bmatrix}$$

Note: Be careful, `linalg.solve` will not work.

Example 2.6.8. Write down the Python script to solve the following eigenvalue problem:

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \end{bmatrix} \mathbf{v} = \lambda \mathbf{v}$$

Sample Solution

```
from scipy import linalg
A = np.array([[-2,11],[17,-19]])
eigenvalues, eigenvectors = linalg.eig(A)
print("/\=",eigenvalues)
print("X=",eigenvectors)
```

There are a few matrix equations from linear control theory, signal processing, filtering, model reduction, image restoration, decoupling techniques for ordinary and partial differential equations below and the respective solvers in Python are listed.

Sylvester equations:

$$\begin{aligned} &AX + BX = C \\ &\texttt{linalg.solve\_sylvester(A, B, C)} \end{aligned} \tag{2.4}$$

are solved using the Bartels-Stewart algorithm.

A continuous-time algebraic Riccati equation (CARE):

$$\begin{aligned} &XA + A^H X - XBR^{-1}B^H X + Q = 0 \\ &\texttt{linalg.solve\_continuous\_are(A, B, Q, R[, E, S, ...])} \end{aligned} \tag{2.5}$$

A discrete-time algebraic Riccati equation (DARE):

$$\begin{aligned} &A^H XA - X - (A^H XB)(R + B^H XB)(B^H XA) + Q = 0 \\ &\texttt{linalg.solve\_discrete\_are(A, B, Q, R[, E, S, balanced])} \end{aligned} \tag{2.6}$$

A continuous-time Lyapunov equation:

$$AX + XA^H = Q \qquad \texttt{linalg.solve\_continuous\_lyapunov(A, Q)} \tag{2.7}$$

A discrete-time Lyapunov equation:

$$AXA^H - X + Q = 0 \qquad \texttt{linalg.solve\_discrete\_lyapunov(A, Q[, method])} \tag{2.8}$$

For square matrices, it is possible to ask more general matrix equation problem such as a 'quadratic equation version for square matrix':

$$AX^2 + BX + C = 0$$

where X, A, B, C, 0 are all $n \times n$ matrices. For example,

$$\begin{bmatrix} -2 & 11 \\ 17 & -19 \end{bmatrix} X^2 + \begin{bmatrix} 19 & 1 \\ 3 & 2 \end{bmatrix} X + \begin{bmatrix} 0 & 13 \\ -13 & 13 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

where X is a 2×2 matrix.

Many of the analytic functions from calculus such as exponential and trigonometric functions can have a 'square matrix version' based on the eigenvalue problem (2.2). An analytic function f has a Taylor series of the form

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k,$$

the corresponding **matrix function** is defined as

$$f(A) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} A^k.$$

The exponential, logarithm, trigonometric and hyperbolic functions can be generalised to become matrix functions calculated by `linalg.expm`, `linalg.logm`, `linalg.sinm`, `linalg.cosm`, `linalg.tanm`, `linalg.sinhm`, `linalg.coshm` and `linalg.tanhm`.

A general **nonlinear matrix problem** has the form:

$$f(X) = 0$$

where X and 0 are $n \times n$ square matrices. There is no simple / unified solution technique to this problem.

Example 2.6.9 (Final Exam Sept 2021 during MCO, Q1(d))**.** Given a 2×2 matrix

$$A = \begin{bmatrix} 1 & -0.1 \\ 0.1 & 1 \end{bmatrix}.$$

Let X be a 2×2 matrix with entries x_{ij}, $i, j = 1, 2$. You are investigating the difference between the matrix exponential function

$$\exp^{[m]}(X) = I_2 + X + \frac{1}{2!}X^2 + \frac{1}{3!}X^3 + \frac{1}{4!}X^4 + \cdots + \frac{1}{k!}X^k + \cdots$$

and the elementwise exponential function

$$\exp(X) = \begin{bmatrix} e^{x_{11}} & e^{x_{12}} \\ e^{x_{21}} & e^{x_{22}} \end{bmatrix}.$$

(i) Write down the Python commands for calculating $\exp^{[m]}(A)$ and $\exp(A)$. Run the Python commands and write down the output of the commands. Then, write down the difference $\exp^{[m]}(A) - \exp(A)$. (1 mark)

Solution: The Python commands are respectively

- $\exp^{[m]}(A)$: linalg.expm(A) ..... [0.2 mark]
- $\exp(A)$: np.exp(A) ..... [0.2 mark]

The outputs are respectively ..... [0.2+0.2=0.4 mark]

```
[[ 2.70470174 -0.27137536]
 [ 0.27137536  2.70470174]]
[[2.71828183 0.90483742]
 [1.10517092 2.71828183]]
```

and the difference is ..... [0.2 mark]

```
[[-0.01358009 -1.17621278]
 [-0.83379556 -0.01358009]]
```

(ii) Write down the Python command to find the difference

$$\exp^{[m]}(A) - I_2 - A - \frac{1}{2!}A^2 - \frac{1}{3!}A^3$$

and write down the difference. (1 mark)

Solution: The Python command to find the difference is ..... [0.8 mark]

```
linalg.expm(A) - np.eye(2) - A - 0.5*A@A - 1/6*A@A@A
```

and the output is ..... [0.2 mark]

```
[[ 0.04803508 -0.02154203]
 [ 0.02154203  0.04803508]]
```

§2.7 Loading, Saving Array and Data Processing

In the real-world, array data are stored in a computer file or saved to a computer file in a certain format. In this section, we will investigate how to store the n-D array that we have discussed in earlier sections and the proceed to discuss how to load and save *structured array*, which closely resembles the *data frame* structure used in Python's famous data processing library `Pandas` (McKinney & PyData Development Team 2019).

§2.7.1 Saving and Loading Array Data

Real-world arrays can be very large. Some of them are stored every second, financial and astronomical data are typical examples. Some of them are large matrices generated from physical models and they will be stored in a special format for fast processing. In this section, we will only introduce the arrays that are less than 2G which can be easily loaded into computer memory and save to a computer file.

The following is a list of array loading and saving functions supported by Numpy (`https://docs.scipy.org/doc/numpy/reference/routines.io.html`) and Scipy (with some external library supports, see `https://docs.scipy.org/doc/scipy/reference/io.html`):

- Text IO: Only 1-D and 2-D arrays could be handled.

  - `np.loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes')`: Load data from a text file.
  - `np.savetxt(fname, A, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ', encoding=None)`: Save an array to a text file.
  - `np.genfromtxt(fname[, dtype, comments, ...])`: Load data from a text file, with missing values handled as specified.

- Serialised IO:
  - `np.load(file, mmap_mode=None, allow_pickle=True, fix_imports=True, encoding='ASCII')`: Load arrays or pickled objects from .npy, .npz or pickled files.
  - `np.save(file, arr, allow_pickle=True, fix_imports=True)`: Save an array to a binary file in Numpy .npy format.
  - `np.savez(file, *args, **kwds)` (and `savez_compressed(file, *args, **kwds)`): Save several arrays into a single file in uncompressed (and compressed) .npz format.

- Raw Binary IO:
  - `np.fromfile(file[, dtype, count, sep])`: Construct an array from data in a text or binary file.
  - `ndarray.tofile(fid[, sep, format])`: Write array to a file as text or binary (default).

- MATLAB® IO: Newer MATLAB files are specialised HDF5 formats while the older MATLAB files required `Matio` library.
  - `scipy.io.loadmat(filename[, mdict, appendmat])`: Load MATLAB file.
  - `scipy.io.savemat(filename, mdict[, appendmat, ···])`: Save a dictionary of names and arrays into a MATLAB-style .mat file.
  - `scipy.io.whosmat(filename[, appendmat])`: List variables inside a MATLAB file.

- Matrix Market files:
  - `scipy.io.mminfo(source)`: Return size and storage parameters from Matrix Market file-like 'source'.
  - `scipy.io.mmread(source)`: Reads the contents of a Matrix Market file-like 'source' into a matrix.
  - `scipy.io.mmwrite(target, A[, comment, field, ···])`: Writes a dense or a sparse array `A` to Matrix Market file-like target.

- Harwell-Boeing files:
  - `scipy.io.hb_read(path_or_open_file)`: Read HB-format file.
  - `scipy.io.hb_write(path_or_open_file, m[, hb_info])`: Write HB-format file.

- Unformatted Fortran files:
  - `scipy.io.FortranFile(filename[, mode, header_dtype])`: A file object for unformatted sequential files from Fortran code.

- HDF5 IO: `h5py` module (`https://www.h5py.org/`) is required.
  - Loading data:

```
with h5py.File('name-of-file.h5', 'r') as hf:
    data = hf['name-of-dataset'][:]
```

– Saving data:

```
with h5py.File('name-of-file.h5', 'w') as hf:
    hf.create_dataset("name-of-dataset",  data=data_to_write)
```

- netCDF IO: `netCDF4` module is required. (`http://unidata.github.io/netcdf4-python/`) is required. This format an extension of HDF5 and is used in Generic Earth Observation Metadata Standard to store satellite data (see `http://avdc.gsfc.nasa.gov/index.php?site=1178067684`)
  - `scipy.io.netcdf_file(filename[, mode, mmap, version, ...])`: A file object for NetCDF data.
  - `scipy.io.netcdf_variable(data, typecode, size, shape, ...)`: A data object for the netcdf module.
- Wav sound files: Unable to handle wav files with 24-bit data.
  - `scipy.io.wavfile.read(filename[, mmap])`: Open a WAV file
  - `scipy.io.wavfile.write(filename, rate, data)`: Write a numpy array as a WAV file.
  - Common data types:

WAV format	Min	Max	Numpy dtype
32-bit floating-point	-1.0	+1.0	float32
32-bit PCM	-2147483648	+2147483647	int32
16-bit PCM	-32768	+32767	int16
8-bit PCM	0	255	uint8

I have been thinking of adding some real-world data for discussion but the smallest is quite large (more than thousand rows), e.g., `https://support.spatialkey.com/spatialkey-sample-csv-data/` and `https://github.com/BenjiKCF/Neural-Network-with-Financial-Time-Series-Data/tree/master/data`. So I will just explore the simpler array data found in earlier sections.

Example 2.7.1 (Final Exam Sept 2013, Q2(b))**.** Write a Python script that perform the following actions:

- Load an array of numbers from a file named `data.txt` and,
- Save the output of ab/n in `result.txt` where
  - a is the largest value in the array,
  - b is the smallest value in the array, and
  - n is the total number of elements in the array. (9 marks)

Sample Solution

```
1 import numpy as np
2 arr = np.loadtxt("data.txt") # each row must have same number of values
3 a = arr.max()
4 b = arr.min()
5 n = arr.size
6 print(f"a={a},b={b},n={n}")
7 fp = open("result.txt","wt")
8 fp.write("{result}\n".format(result=a*b/n))
9 fp.close()
```

Example 2.7.2. Write the Python command to store the array `A` from Example 2.2.2 into a text file `array.txt` and Numpy binary file `array.npy` and used notepad (or other editors) to open these file. What do you observe?

§2.7.2 Array of Structures

A *structured array* or an *array of structures* is an n-D array whose datatype is a composition of sequence data with fields (`https://jakevdp.github.io/PythonDataScienceHandbook/02.09-structured-data-numpy.html`). These type of arrays are found everywhere from business records to government records.

The way to construct simple structured array in Python/Numpy is

```
S = np.array([(1,2.0),(3,4.0)], dtype=[('foo', 'i8'), ('bar', 'f4')])
```

According to `np.dtype` document, a character code is used to identify the general kind of data: `b` for boolean, `i` for signed integer, `u` for unsigned integer, `f` for floating-point, `c` for complex floating-point, `m` for timedelta, `M` for datetime, `O` for Python object, `S` for (byte-)string, `U` for Unicode and `V` for void. The integer denotes the number of bytes.

Example 2.7.3 (Final Exam Sept 2015, Q5(c))**.** Based on the following table, create a vector of structures called `student` that when displayed has the following values:

```
>> student
student
     1x3  struct array with fields:
           name
           id_no
           quiz
```

name	id_no	quiz
Mickey	M_001	10.5
Minnie	M_002	8.5
Donald	M_003	6.5

(6 marks)

Solution: The MATLAB command is shown below:

```
student = struct('name',{'Mickey','Minnie','Donald'},
  'id_no',{'M_001','M_002','M_003'},'quiz',{10.5,8.5,6.5})
```

The Python commands are shown below:

```
1 import numpy as np
2 student = np.array([('Mickey', 'M_001', 10.5),
3     ('Minnie', 'M_002', 8.5), ('Donald', 'M_003', 6.5)],
4     dtype=[('name','U20'), ('id_no', 'U5'), ('quiz', '<f8')])
```

§2.7.3 DataFrame Handling

A less complicated "structured array" is the *DataFrame* which is a 2-D array with column names and an index. Python pandas library provides the extension to Numpy to handle DataFrame and the documentation is available from `https://pandas.pydata.org/`.

In what follows, we will illustrate how to use DataFrame to merge the course works given by two lecturers.

Example 2.7.4. In a university, the basic numerical data is the handling of student results. A sample data from `https://sites.google.com/site/liewhowhui/scicomp/notes/result2018.csv` is listed below.

```
PROGRAMME,Name,T1Q1a,T1Q1b,T1Q1c1,T1Q1c2,T1Q1d,T1Q2a,T1Q2b1,T1Q2b2,T2Q1a1,T2Q1a2,T2Q1a3,T2Q1b,T2Q2a,T2Q
AM,Student1,0,0,0.3,0,1,6.2,0.6,2.5,0.5,0,0.5,2.5,0,4,1,1.4,0.6,12,19,9,20,,5
AM,Student2,0,2,0.9,1,1.6,6.3,0,2.4,0.5,1.7,2.9,1.4,0,3,1,1.4,0.6,13,14,11,17,9,
AM,Student3,1.3,1.5,0.3,0,1.6,6.5,0.6,2.5,0.5,0.5,0.5,4.6,0,1.5,1,1.4,0.6,16,14.5,17,15,6,
AM,Student4,1.8,0.6,0.9,0,2,5.5,0.1,2.4,0.5,2,3,5.4,0.6,3.5,1,1.4,0.6,16,11.5,16,20,,7
AM,Student5,1.8,2,1,2.2,2,6.1,0.5,1.7,0.5,2,3,4.8,0,4,1,1.4,0.6,20,19.5,16,20,13,
AM,Student6,1,2,0.6,0.8,1,3,0.5,1,0.5,2,3,1.3,1,1,1,1.4,0.6,15,7,11,17,7,
AM,Student7,1.7,0.5,1,2.2,1.8,5.5,1,2.5,0,1.2,1.7,5.7,0,3,1,1.4,0.6,20,8.5,6,15,,13
AM,Student8,2,2,1,2.5,2,5.7,1,2.5,0.5,2,3,6.5,0,4,1,1.4,0.6,20,20,19.5,20,,20
AM,Student9,2,2,0.9,2.1,1,4.2,1,2.1,0.5,2,3,6.4,0.7,4,1,1.4,0.6,18,19.5,9,18,,14.5
AM,StudentA,1.4,2,0.9,3,2,6.5,1,2,0.5,2,3,4.2,0.8,3,1,1.4,0.6,18,18,19,15,12,
AM,StudentB,1.3,0.9,1,0,1.1,6.5,1,1.5,0.5,2,3,5.8,0.3,1.6,1,1.4,0.6,14,13.5,9,9.5,,13
AM,StudentC,1.7,2,0.9,0.4,1.4,6.3,0.6,2.1,0.5,2,3,6.3,0.2,4,1,1.4,0.6,20,10.5,7,18,19,
AM,StudentD,2,1.5,0.9,2.4,2,6.5,1,0.5,0.5,1.9,3,3,0,3,1,1.4,0.6,18,14.5,15,20,1,
AM,StudentE,1,2,0.9,2,2,6.5,1,1.5,0.5,2,3,6.5,0.6,4,1,1.4,0.6,16,19.5,19.5,20,,18
AM,StudentF,1.4,2,0.4,1,1,5.9,1,2,0.5,1.7,3,3.3,0,4,1,1.4,0.5,17,15.5,16,15,13,
```

The data has "strings" and "numbers", the best Python module to handle this is `pandas`. The following is a Python script to calculate the totals and course outcomes (CO) of the test 1, test 2 and final results.

```
import pandas as pd
data = pd.read_csv("result2018.csv")
print(data.dtypes)   # OK similar to Numpy
dv = data.as_matrix()  # Convert from dataframe to matrix
dv = dv[:,2:].astype('float')  # Take only the numbers
data['T1Q1'] = data[data.columns[2:7]].sum(axis=1)
data['T1Q2'] = data[data.columns[7:10]].sum(axis=1)
data['T2Q1'] = data[data.columns[10:14]].sum(axis=1)
data['T2Q2'] = data[data.columns[14:16]].sum(axis=1)
data['T2Q3'] = data[data.columns[16:19]].sum(axis=1)
idx = data[['T1Q1','T1Q2','T2Q1','T2Q2','T2Q3']].sum(axis=1)<20.0
data[idx]['Name']
# Weights for T1Q1, T1Q2, T2Q1, T2Q2, T2Q3 = 10, 10, 12, 5, 3
# True Weights for Final Q1, ..., Q6 = 20 marks x 0.6 = 12 marks
CO1 = (data['T1Q1'] + 0.6*data['FEQ1'])/22*100 # T1Q1=10%, FEQ1=12%
CO2 = (data['T1Q2'] + 0.6*data['FEQ2'])/22*100 # T1Q2=10%, FEQ2=12%
CO3 = (data['T2Q1'] + 0.6*data['FEQ3'])/24*100 # T2Q1=12%, FEQ3=12%
CO4 = (data['T2Q2'] + 0.6*data['FEQ4'])/17*100 # T2Q2=5%, FEQ3=12%
CO5 = (data['T2Q3'] + 0.6*data[['FEQ5','FEQ6']].sum(axis=1))/15*100
```

Example 2.7.5. Consider the course work results of two lecturers below.

```
StudentID,T1Q1,T1Q2
1805764,5,8.5
1903172,6,9.0
1904690,2,9.5
1805707,8,9.0
1904324,4,9.0
1900585,8,8.5
1900059,8,10.0
1805313,8,10.0
1903374,5,7.5
```

```
StudentID,T2Q1,T2Q2
1805764,8.5,0.5
1903374,5.4,2.5999999999999996
1904324,11.799999999999999,6.0
1903172,9.5,7.1
1805313,9.1,8.8
1805707,9.7,8.8
1904690,9.899999999999999,8.2
1900059,14.2,9.2
1900585,12.5,7.2
```

(a) Explain why does some numbers have so many 9s in the decimal part?

(b) After the data has been read by using `pd.read_csv`, explain how should the DataFrame be merged to obtain a total result.

(c) Round the total marks to integer and explain the problem with using `np.round`.

References

Haenel, V., Gouillart, E. & Varoquaux, G. (2015), Python scientific lecture notes release 2013.2 beta (euroscipy 2013). `https://www.scipy-lectures.org/_downloads/PythonScientific-simple.pdf`.

Kiusalaas, J. (2005), *Numerical Methods in Engineering with Python*, Cambridge University Press.

Langtangen, H. P. (2004), *Python Scripting for Computational Science*, Springer Verlag.

McKinney, W. & PyData Development Team (2019), *Pandas: Powerful Python Data Analysis Toolkit Release 0.24.1.*

Oliphant, T. (2006), Guide to numpy. `http://web.mit.edu/dvp/Public/numpybook.pdf`.