[Not for Exam]

==========

CS1302 - Lecture X Numerical Analysis Numpy (Numerical_Analysis.ipynb)

(Dr Helena WONG)

==========

Numpy

1. Monte Carlo Simulation

finding the value of *pi* (3.14..)

2. Linear Algebra

Solve for x and y :

$$\begin{cases} 4x + 8y = 20 & \text{——①} \\ 2x + 6y = 8 & \text{——②} \end{cases}$$

2.1 Creating numpy arrays

2.2 Operating on numpy arrays

2.3 Universal functions

==========

```
import math
import random
```

==========

Introduction of NumPy (Important for Data Science and scientific applications etc.)

https://www.w3schools.com/python/numpy/numpy_intro.asp

NumPy is for working with arrays (like a list**, tuple)**

NumPy aims to provide an array object called **ndarray, with many functions**

the items have the same type (e.g. all are integer, all are double) "efficiency"

NumPy can handle them quick because of fixed storage size

3D array: 2 pages x 3 rows x 7 columns

	A	B	C	D	E	F	G	H
1		Sun	Mon	Tue	Wed	Thu	Fri	Sat
2	Shop 1	94.7%	77.6%	17.7%	17.3%	5.2%	43.3%	21.7%
3	Shop 2	39.1%	47.5%	60.8%	26.2%	36.4%	60.7%	67.1%
4	Shop 3	1.8%	34.8%	73.5%	19.8%	26.2%	38.0%	78.6%
5								
6								

Jan | Feb

Numpy: Handle very big 3D array!: 200 pages x 20000 rows x 700 columns

```
import numpy as np
a = np.zeros(7)
a, type(a), len(a), type(a[0]), a[0] # (array([0., 0., 0., 0., 0., 0., 0.]), numpy.ndarray, 7, numpy.float64, 0.0)

np.random.seed(0)
a = np.random.random(7) # seven random numbers [0 to 1.0)
a, type(a), len(a), type(a[0]), a[0]

a = np.random.randint(10000,200000,2000000) # very quick: a lot of random numbers (10000, 200000)
a, type(a), len(a), type(a[0]), a[0]
```

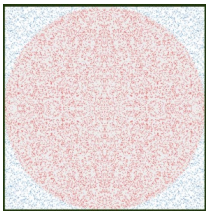
```
import random
a=list(random.randint(10000,200000) for i in range(2000000)) # ~ Some seconds
type(a), len(a), type(a[0]), a[0]
print(*a[:3], "..", *a[-3:])
print(type(a), len(a), type(a[0]), a[0])

a = np.random.randint(1, 6, 10) # ten random numbers
b = a**2 # broadcasting
a, b, a>3
Output:
(array([3, 4, 3, 2, 5, 4, 4, 4, 2, 4]),
 array([ 9, 16,  9,  4, 25, 16, 16, 16,  4, 16]),
 array([False,  True, False, False,  True,  True,  True,  True, False, True]))

a = np.random.randint(1,6,7) # 1D array: size is 7
b = np.random.randint(1,6,(7,2)) #2D array: shape is 7x2
a,b
Output:
(array([3, 2, 4, 3, 2, 1, 5]),
 array([[1, 2],
        [4, 3],
        [4, 4],
        [4, 5],
        [4, 2],
        [4, 5],
        [1, 5]]))

a = np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24])
a = a.reshape((2,3,4)) # 3D Array: 2x3x4 (2 pages, each 3 rows; 4 columns per row)
print(a)
print(a.sum())
print(a.sum(axis=0)) # sum across pages
print(a.sum(axis=1)) # sum across rows
print(a.sum(axis=2)) # sum across columns
print(a.sum(axis=-1))
==========================================
```

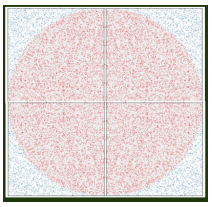
If we uniformly randomly pick a point in a square.
What is the chance it is in the inscribed circle, i.e., the biggest circle inside the square?



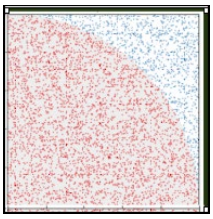
The chance is the area of the circle divided by the area of the square. Suppose the square has length ℓ, then the chance is

$$\frac{\pi(\ell/2)^2}{(\ell)^2} = \frac{\pi}{4}$$

independent of the length ℓ.



Show this image in a mark-down cell:
<img src="https://upload.wikimedia.org/wikipedia/commons/8/84/Pi_30K.gif" alt="Pi 30K.gif">



==========

approximate_pi

```
import random, math
def approximate_pi(n): # e.g. n is 10000 times

    return result # e.g. result is ~3.1

print("Approximate: ", approximate_pi(10000))
print("Ground truth: ", math.pi)
```

```
import random, math
def approximate_pi(n): # e.g. n is 10000 times
    hit = 0
    for i in range(n): # run 10000 times
        x=random.random()
        y=random.random()
        if x*x + y*y < 1:
            hit += 1
    result = 4*(hit/n)
    return result # e.g. result is 3.12

print("Approximate: ", approximate_pi(10000))
print("Ground truth: ", math.pi)
```

==========

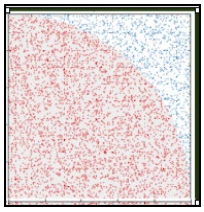
np_approximate_pi

```
import numpy as np

def np_approximate_pi(n):
    in_circle = (np.random.random((n,2))**2).sum(axis=-1) < 1
    mean = 4 * in_circle.mean()
    std = 4 * in_circle.std() / n**0.5
    return np.array([mean - 2*std, mean + 2*std])

interval = np_approximate_pi(int(1e7))
print(f'''95%-confidence interval: {interval}
Estimate: {interval.mean():.4f} ± {(interval[1]-interval[0])/2:.4f}
Ground truth: {math.pi}''')
```

Re-written for explanation



```
import numpy as np
import math
n = 20 # or 1000, number of points

# in_circle = (np.random.random((n,2))**2).sum(axis=-1) < 1
points = np.random.random((n,2)) # x and y of 1000 points
sq_dist_from_center = (points**2).sum(axis=1)
in_circle = sq_dist_from_center < 1

mean = 4 * in_circle.mean()
std = 4 * in_circle.std() / n**0.5
interval = np.array([mean - 2*std, mean + 2*std])

print("95%-confidence interval: {:.5f} - {:.5f}".format(
    interval[0], interval[1]))

err = (interval[1]-interval[0])/2
print("Estimate: {:.5f} ± {:.5f}".format(interval.mean(), err))
print("Ground truth: ", math.pi)
```

```
print(*points[:2], "..", *points[-2:])
print(*sq_dist_from_center[:2], "..", *sq_dist_from_center[-2:])
print(*in_circle[:5], "..", *in_circle[-5:])
```

==========

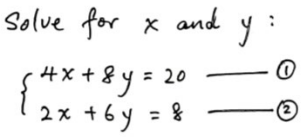
```
def solve_linear_equation(a,b): # ax=b has a solution or None?
    if a!=0:
        return b/a
    else:
        return None

import ipywidgets as widgets
@widgets.interact(a=(0,5,1),b=(0,5,1))
def linear_equation_solver(a=2, b=3):
    print(f'''linear equation: {a}x = {b}
        solution: x = {solve_linear_equation(a,b)}''')
```

==========

What if we have 2 variables x and y?

Using **numpy** to store the matrix of the coefficients




```
A = np.array([[4.,8],[2,6]])
b = np.array([20.,8])
A, b
```

Getting values:

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

==========

numpy implements different data types with different storage sizes;

note the range of values

```
np.int64(2**62) # OK
np.int64(2**63) # OverflowError
```

```
a1 = 2**62
a2 = 2**63
print(a1, a2) # OK

np.int64(a1) # OK
np.int64(a2) # OverflowError
```

==========

Conversion between data types:

```
A1 = np.int64([1, 2, 3])
A2 = A1.astype(np.float32)
A3 = A2.astype(np.double)
print(A1, A1.dtype, type(A1)) # [1 2 3] int64 <class 'numpy.ndarray'>
print(A2, A2.dtype, type(A2)) # [1. 2. 3.] float32 <class 'numpy.ndarray'>
print(A3, A3.dtype, type(A3)) # [1. 2. 3.] float64 <class 'numpy.ndarray'>

A1[0] = 4.5
print(A1) #[4 2 3]
print(np.array([int(1), float(1)])) #[1. 1.] will be all floating point numbers
```

==========

```
heterogeneous_np_array = np.array([0, 1, 2, 'a', 'b', 'c'])
heterogeneous_np_array # array(['0', '1', '2', 'a', 'b', 'c'], dtype='<U21')
```

The dtype '<U21' stands for unicode strings with 21 characters or less.

https://www.gormanalysis.com/blog/python-numpy-for-your-grandma-2-2-numpy-array-basics/

```
heterogeneous_np_array = np.array([0, 1, 2, 'a', 'b', 'c'], dtype=object)
heterogeneous_np_array # array(['0', '1', '2', 'a', 'b', 'c'], dtype=object)
```

==========

numpy provides many functions to create an array:

```
# use online python tutor: import numpy as np
a1 = np.zeros(0), np.zeros(1), np.zeros((2,3,4))    # Dimension can be higher than 2
a2 = np.ones(0, dtype=int), np.ones((2,3,4), dtype=int)    # initialize values to int 1
a3 = np.eye(0), np.eye(1), np.eye(2), np.eye(3)    # identity matrices
a4 = np.diag(range(1)), np.diag(range(2)), np.diag(range(4))    # diagonal matrices
a5 = np.empty(0), np.empty((2,3,4), dtype=int)    # create array faster without initialization
```

==========

numpy also provides functions to build an array using rules:

```
# https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

# arange: like range but allow non-integer parameters
# linspace: (linear spaced) specify number of points instead of step

# use online python tutor: import numpy as np
a1 = np.arange(5)
a2 = np.arange(4,5)
a3 = np.arange(4.5,5.5,0.5)

a4 = np.linspace(4,5)
a5 = np.linspace(4,5,6)
a6 = np.linspace(4,5,11)

# fromfunction: initialize using a function
a7 = np.fromfunction(lambda i, j: i * j, (3,4))

print(a4.size)
print(len(a4))
print(a7.ndim) # number of dimensions
```

reshape and flatten

```
array1 = np.arange(2*3)
array2 = np.arange(2*3).reshape(3,2)
array3 = np.arange(2*3).reshape(2,3)
array4 = np.arange(2*3).reshape(2,3).flatten()
```

==========

```
import numpy as np

def print_array_entries_line_by_line(array):
    # print(array)
    for i in array.flatten():
        print(i)

print_array_entries_line_by_line(np.arange(4*3).reshape(2,2,3)) # print 0-11 in 12 lines
```

==========

Matrix Multiplications

https://www.mathsisfun.com/algebra/matrix-multiplying.html

```
# use online python tutor: import numpy as np
A1 = np.array([[1,2,3],[4,5,6]])
A2 = np.array([[7,8],[9,10],[11,12]])
C = np.array([[999,64],[139,154]])

D = np.matmul(A1,A2)
D1 = D>150, D==C
```

```
# using @
A1 = np.array([[1,2,3],[4,5,6]])
A2 = np.array([[7,8],[9,10],[11,12]])
D3 = A1@A2
```

==========

 # .all()

```
# use online python tutor: import numpy as np
D = np.array([[58,64],[139,154]])

r1 = (D==np.array([[999,64],[139,154]]))
r2 = (D==np.array([[58,64],[139,154]]))
r3 = (D==np.array([[999,64],[139,154]])).all() #False
r4 = (D==np.array([[58,64],[139,154]])).all() #True
```

==========

combine A and b to form augmented matrix

Solve for x and y :

$$\begin{cases} 4x + 8y = 20 \quad ① \\ 2x + 6y = 8 \quad ② \end{cases}$$

```
A = np.array([[4.,8],[2,6]])
b = np.array([20.,8])
```

```
# use online python tutor: import numpy as np
A = np.array([[4.,8],[2,6]])
b = np.array([20.,8])

C = np.block([A,b.reshape(-1,1)])    # reshape to ensure same ndim

# about reshape:
b = np.array([20.,8])
b1 = b.reshape(2,1)
b2 = b.reshape(-1,1) # the unspecified value is inferred
```

==========

To stack an array along different axes:

```
# use online python tutor: import numpy as np
array = np.array([[0,1,2],[3,4,5]])
r1 = np.hstack((array,array))      # stack along the first axis
r2 = np.vstack((array,array))                    # second axis
r3 = np.concatenate((array,array), axis=-1)       # last axis, here is same as hstack
r4 = np.stack((array,array), axis=0)                # new axis
```

==========

To divide all the coefficients by 2

```
C = np.array([[2.,2,1],[0,2,1]])
D = C / 2
```

==========

*** and broadcasting**

Using *

```
# use online python tutor: import numpy as np
x33 = np.array([[0,0,0],[1,1,1],[2,2,2]])
y33 = np.array([[0,1,2],[0,1,2],[0,1,2]])
x43 = np.array([[0,0,0],[1,1,1],[2,2,2],[3,3,3]])
y43 = np.array([[0,1,2],[0,1,2],[0,1,2],[0,1,2]])

z33 = x33*y33
# z1 = x33*y43 # error
# z2 = x43*y33 # error
```

Using * and broadcasting (stretch the axis of size 1 up to match the corresponding axis in other arrays.)

```
# use online python tutor: import numpy as np
x41 = np.array([[0,1,2,3]])
y14 = x41.reshape(4,1)
z1 = x41*y14
z2 = y14*x41
```

==========

subtract the second row of the coefficients from the first row:

```
# use online python tutor: import numpy as np
C = np.array([[2.,2,1],[0,2,1]])
D = C / 2

D_backup=D.copy()

D[0,:] = D[0,:] - D[1,:]
```

==========

Get last column

```
x = D[:,-1]
```

==========

Indexing and slicing: one-dimension array

```
a = np.arange(10)
a_slice = a[2:7:2]
a_slice[1] = 99
# the results: [0 1 2 3 99 5 6 7 8 9] and [2 99 6]
```

```
#Compare it with List:
list1 = [*range(10)]
list2 = list1[2:7:2]
list2[1] = 99
```

==========

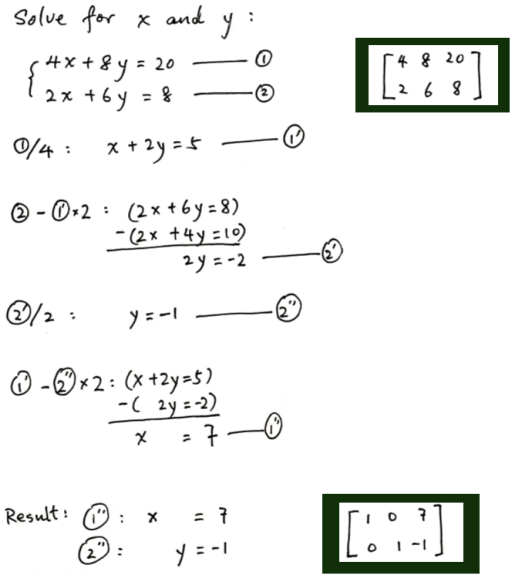
Indexing and slicing: two-dimension array

```
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
a_slice = a[1:]
a_slice[0][1] = 99

# the results:
[[1 2 3]
 [3 99 5]
 [4 5 6]]

[[3 99 5]
 [4 5 6]]
```

==========



solve_2_by_2_system

```
def solve_2_by_2_system(A,b): #suppose the equations are 4x+8y = 20 and 2x+6y = 8
    '''Returns the unique solution of the linear system, if exists,
    else returns None.'''
    C = np.hstack((A,b.reshape(-1,1)))
    if C[0,0] == 0: C = C[(1,0),:] #swap the two rows
    if C[0,0] == 0: return None
    C[0,:] = C[0,:] / C[0,0]
    C[1,:] = C[1,:] - C[0,:] * C[1,0]
    if C[1,1] == 0: return None
    C[1,:] = C[1,:] / C[1,1]
    C[0,:] = C[0,:] - C[1,:] * C[0,1]
    return C[:,-1] # -1 means last column, that will be 7 and -1

A0 = np.array([[4,8],[2,6]])
b0 = np.array([20,8])
print(solve_2_by_2_system(A0,b0))
```

Steps

(1) 4x+8y = 20

(2) 2x+6y = 8

Change (1): divide (1) by 4 => x+2y=5

*Change (2): (2) - (1)*2 => i.e. 2x+6y=8 minus 2x+4y=10 => 2y=-2*

Change (2): (2)/2 => y=-1

*Change (1): (1) - (2)*2 => i.e. x+2y=5 minus 2y=-2 => x=7*

Solved!

==========

Universal functions

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion

(1) familiar mathematical functions, such as np.exp etc.

```
a = np.array([[1, 2, 3],[4,5,6]])
b = np.exp(a)    # Calculate the exponential of given number
```

(2) fromfunction()

Examples of usage:

```
a1 = np.fromfunction(lambda i,j:i, (3,4), dtype=int)
a2 = np.fromfunction(lambda i,j:j, (3,4), dtype=int)
a3 = np.fromfunction(lambda i,j:(i*j), (3,4), dtype=int)
a4 = np.fromfunction(lambda i,j:(i,j), (3,4), dtype=int)
```

Explanation of **`a3 = np.fromfunction(lambda i,j:(i*j), (3,4), dtype=int)`**

Try:

```
f = lambda i,j: (i*j)
x3 = np.fromfunction(f, (3,4), dtype=int)

# From the documentation, fromfunction applies the given function to the two arrays as arguments.
# * is broadcasted
y3 = f(np.array([[0,0,0,0],[1,1,1,1],[2,2,2,2]]),np.array([[0,1,2,3],[0,1,2,3],[0,1,2,3]]))
```

Explanation of **`a4 = np.fromfunction(lambda i,j:(i,j), (2,3), dtype=int)`**

Try:

```
f = lambda i,j: (i,j)
x4 = np.fromfunction(f, (3,4), dtype=int)

# f takes in two arrays i and j, and returns a tuple containing the two arrays (no broadcasting)
y4 = f(np.array([[0,0,0,0],[1,1,1,1],[2,2,2,2]]),np.array([[0,1,2,3],[0,1,2,3],[0,1,2,3]]))
```

```
def k(i,j):
    # print(i)
    # print(j)
    return i*j

a5 = np.fromfunction(k,(3,4), dtype=int)
```