# CCPS109
# Computer Science I
# L6-7

**Lecturer: Nhan Tran**
**nhantran@Ryerson.ca**

# Agenda

Apologies, could not get narration to work. Some thing about windows hardware permission setup.

Will continue try to figure it out for the next lecture, I will do something.

If you have question send me email and I will try to clarify it next week.

Start on the 13 take home labs!!!

# Agenda

**Announcement**

## Lecture:

numpy

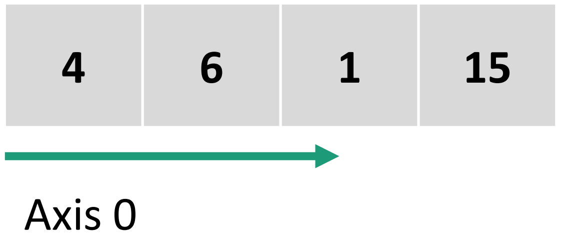
Testing, Debugging,

# Numpy

## Numerical Python

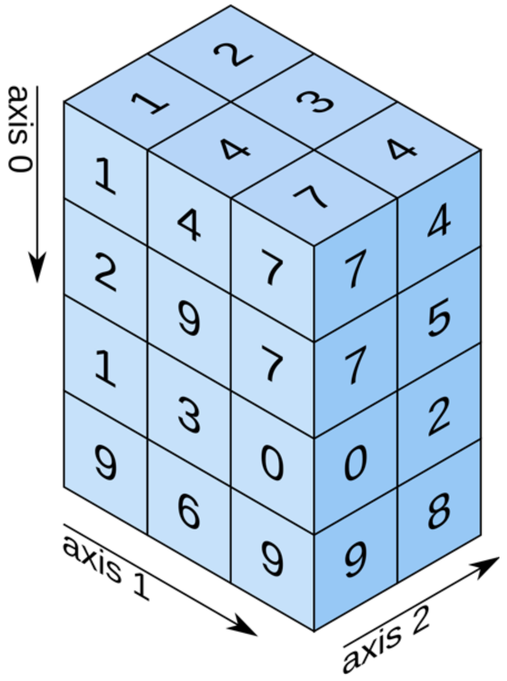
- Homogeneous multidimensional array.
  - Typically tables of numbers
  - Index by tuple of non-negative integers
  - Dimension are referred as **axes**

**1D array**




**2D array**


**3D array**




3d figure: https://miro.medium.com/max/1440/1*Ikn1J6siiiCSk4ivYUhdgw.png

# Numpy

- Data can be in N dimensional array: 0<N<=infinity
- High performance
- Scientific computing and data analysis
- Fast, space-efficient
- Have tools for reading/writing array data to file
- Have standard mathematical function for fact operation

import numpy as np

# Installing Numpy

https://scipy.org/install.html

Once numpy is installed you can import the package and use it:

```
import numpy as np
```

# Numpy Arrays

Creating array/matrix by simply passing a python list or any sequence into the `array` function

```
import numpy as np    #always import numpy
ar=np.array([2,3,4,7])
```

1dimension integer array

| | | | |
|---|---|---|---|
| **2** | **3** | **4** | **7** |

or

```
import numpy as np
L1= [2,3,4,7]
ar=np.array(L1)
```

To access the value, use index just like lists

ar[2]     <=result in 4

# Numpy indexing 1D

ar = 

| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 7 |

slice is the same

ar[0:2]

| | |
|---|---|
| 2 | 3 |

ar[1:]

| | | |
|---|---|---|
| 3 | 4 | 7 |

Try: ar.max() , ar.min(), ar.sum(), prod(), mean(), std()

# arrays

```
import numpy as np
data1 = [6, 7.5, 8, 0, 1]

arr1 = np.array(data1)
 #resulting in float array:
 #array([ 6. ,  7.5,  8. ,  0. ,  1. ])
print(arr1.dtype)
```

#change all values to floating

#Float64

# 2d Array

```
import numpy as np
data2 = [[1, 4, 3, 2], [5, 6, 7, 8]]
arr2 = np.array(data2)
print(arr2)
Output:
array([[1, 4, 3, 2],
       [5, 6, 7, 8]])

print(arr2.ndim)
Output: 2

print(arr2.shape)
Output: (2, 4)
```

```
print(arr2[1][2])    #what is the ouput?
```

# 2d Array

```
import numpy as np
data2 = [[1, 4, 3, 2], [5, 6, 7, 8]]
arr2 = np.array(data2)
print(arr2)
Output:
array([[1, 4, 3, 2],
       [5, 6, 7, 8]])

print(arr2.ndim)
Output: 2

print(arr2.shape)
Output: (2, 4)
```

What is the output when printing the following:

print(arr2[1,2])
#slice
arr2[1:3]
arr2[0:2,0]

arr2.max(axis=0) #column

arr2.max(axis=1) #row

!n: if you want a copy of the slice, must make a copy

Arslice=arr2[1:3].copy()

# 2d indexing

Axis 0
column

Axis 1
row

| | 0 | 1 | 2 |
|---|---|---|---|
| **0** | **0,0** | **0,1** | **0,2** |
| **1** | **1,0** | **1,1** | **1,2** |
| **2** | **2,0** | **2,1** | **2,2** |

# 2d indexing

Axis 0
Column

| | 0 | 1 | 2 |
|---|---|---|---|
| **0** | **0,0** | **0,1** | **0,2** |
| **1** | **1,0** | **1,1** | **1,2** |
| **2** | **2,0** | **2,1** | **2,2** |

Axis 1
Row

Arr[1,2]

# Numpy ndarray method

- .ndim
  - the number of axes (dimensions) of the array.
- .shape
  - the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
- .size
  - the total number of elements of the array. This is equal to the product of the elements of shape.
- .dtype
  - an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types.
  - Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

# Numpy ndarray method

## .itemsize

- the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

## .data

- Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

# Transpose .T

arr

| | |
|---|---|
| **1** | **9** |
| 4 | 2 |
| 5 | 7 |
| 3 | 6 |

arr.T

| | | | |
|---|---|---|---|
| 9 | 2 | 7 | 6 |
| 1 | 4 | 5 | 3 |

# Reshape

Reshape() return a new array with passed in dimensions.

arr

| |
|---|
| 9 |
| 2 |
| 7 |
| 6 |
| 1 |
| 4 |
| 5 |
| 6 |

arr.reshape(2,4)

| | | | |
|---|---|---|---|
| 9 | 2 | 7 | 6 |
| 1 | 4 | 5 | 3 |

# reshape and arrange()

Reshape() return a new array with passed in dimensions.

```
b = np.arange(12).reshape(3,4)    # arange() return an array & takes floating point,
                                    #it is similar to range()
b
>>>
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

# Generating array

```
np.arange(15)
 output>>array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

np.zeros(10)
Output>> array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

np.zeros((3, 6))
Output>>
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

Try:

```
onesarray = np.ones((3,6)
```

# Generating array

```
np.empty((2, 3, 2))
Output>>   #3d array
array([[[  4.94065646e-324,   4.94065646e-324],
        [  3.87491056e-297,   2.46845796e-130],
        [  4.94065646e-324,   4.94065646e-324]],

       [[  1.90723115e+083,   5.73293533e-053],
        [ -2.33568637e+124,  -6.70608105e-012],
        [  4.42786966e+160,   1.27100354e+025]]])
```

DO NOT assume that **np.empty** will return an array of all zeros. It will return uninitialized garbage values at the memory location that was allocated when creating the array.

# Array Creation Functions

| Function | Description |
|---|---|
| array | Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default. |
| asarray | Convert input to ndarray, but do not copy if the input is already an ndarray |
| arange | Like the built-in range but returns an ndarray instead of a list. |
| ones, ones_like | Produce an array of all 1’s with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype. |
| zeros, zeros_like | Like ones and ones_like but producing arrays of 0’s instead |
| empty, empty_like | Create new arrays by allocating new memory, but do not populate with any values like ones and zeros |
| eye, identity | Create a square N x N identity matrix (1’s on the diagonal and 0’s elsewhere) |

# Data type

Specifying data type for array using dtype

```
arr1 = np.array([1, 2, 3], dtype=np.float64)

arr2 = np.array([1, 2, 3], dtype=np.int32)
```

This allow for more control over how data are store in memory or disk whether they are integers, floating point, Boolean, string, or python objects.

Look up: Numpy data types

# 3d array

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

arr3d[0]
Output>>                              #2x3 array

array([[1, 2, 3],
       [4, 5, 6]])
```

!n: for multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions

# 3d array

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3d
Output>>  # 2x2x3 array
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

# Matrix operations

All Matrix operations are support by numpy:

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
print(b)
>>array([0, 1, 2, 3])
c = a-b
print(c)
array([20, 29, 38, 47])
print(b**2)
array([0, 1, 4, 9])
```

# Matrix operations

All Matrix operations are support by numpy:

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
print(b)
>>array([0, 1, 2, 3])
#Try:
c = a+b
print(c)
a+=2          #mutate a by add 2 to each element
              # *= ,
```

# Matrix operations

```
a = np.array( [20,30,40,50] )

print(10*np.sin(a))
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])

print(a<35)
array([ True,  True, False, False])
```

# Operation:

```
A = np.array( [[1,1],
               [0,1]] )

B = np.array( [[2,0],
               [3,4]] )
A * B                         # elementwise product
output>>
      array([[2, 0],
             [0, 4]])
```

# Operation: product

```
A = np.array( [[1,1],
               [0,1]] )
B = np.array( [[2,0],
                [3,4]] )
```

```
A @ B               # matrix product
output>>
        array([[5, 4],
               [3, 4]])
```

```
A.dot(B)               # another matrix product
output>>
      array([[5, 4],
       [3, 4]])
```

# Operation: axis

```
b = np.arange(12).reshape(3,4)
print(b)
Output>>
        array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])

b.sum(axis=0)                    # sum of each column
Output>>   array([12, 15, 18, 21])
```

# Operation: axis

```
b.min(axis=1)                    # min of each row
Output>>
        array([0, 4, 8])

b.cumsum(axis=1)                 # cumulative sum along each row
output>>
        array([[ 0,  1,  3,  6],
               [ 4,  9, 15, 22],
               [ 8, 17, 27, 38]])
```

# Resize

given a is already define:

```
a
output>>
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])

a.resize((2,6))
print(a)
array([[3., 7., 3., 4., 1., 4.],
       [2., 2., 7., 2., 4., 9.]])
```

resize() mutate the current array
Unlike reshape()  which return a new array

# Numpy: Additional resource

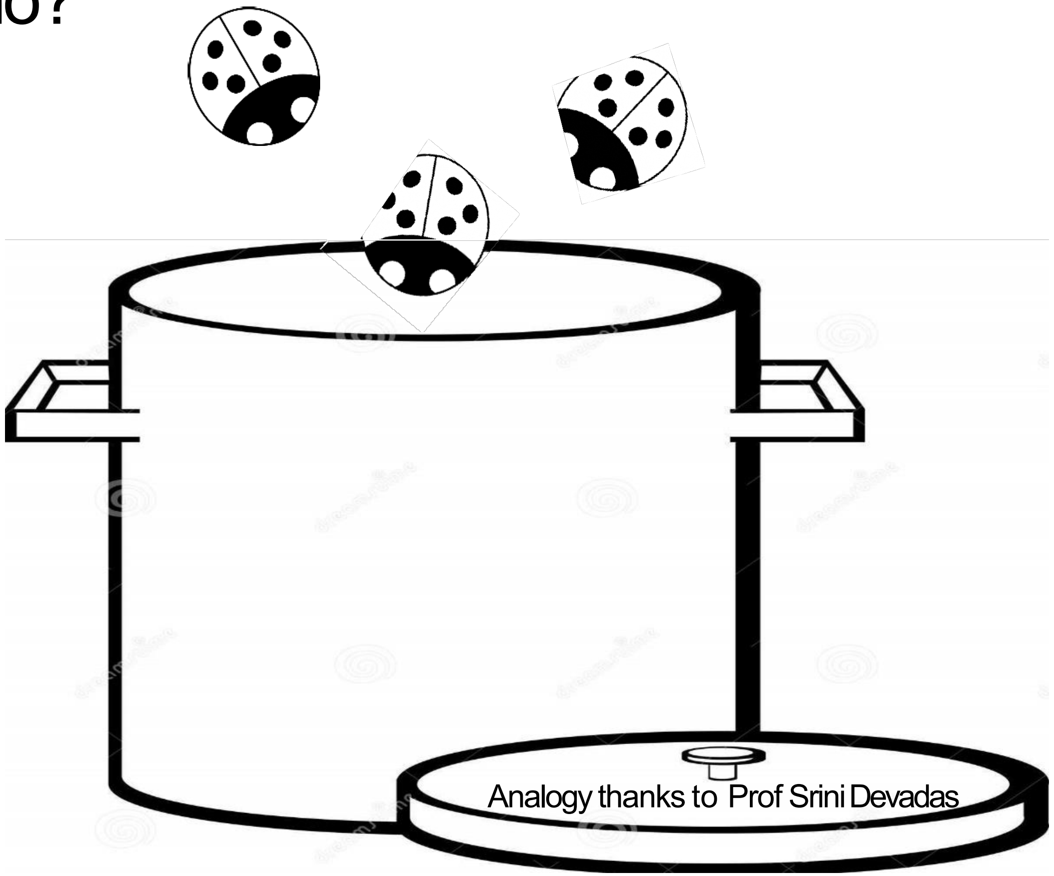
- https://numpy.org/devdocs/user/quickstart.html#
- https://scipy-lectures.org/intro/numpy/operations.html
- https://www.oreilly.com/library/view/python-for-data/9781449323592/ch04.html

# TESTING, DEBUGGING, EXCEPTIONS, ASSERTIONS

# WE AIM FOR HIGH QUALTIY – AN ANALOGY WITH SOUP

You are making soup but bugs keep falling in from the ceiling. What do you do?

- check soup for bugs
  - testing
- keep lid closed
  - defensive programming
- clean kitchen
  - eliminate source of bugs



## DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)

## TESTING / VALIDATION

- **Compare** input/output pairs to specification
- “It’s not working!”
- “How can I break my program?”

## DEBUGGING

- **Study events** leading up to an error
- “Why is it not working?”
- “How can I fix my program?”

# SET YOUSELF UP FOR EASY TESTING AND DEBUGGING

- from the **start**, design code to ease this part
- break program up into **modules** that can be tested and debugged individually
- **document constraints** on modules
  - what do you expect the input to be?
  - what do you expect the output to be?
- **document assumptions** behind code design

# WHEN ARE YOU READY TO TEST?

- ensure **code runs**
  - remove syntax errors
  - remove static semantic errors
  - Python interpreter can usually find these for you
- have a **set of expected results**
  - an input set
  - for each input, the expected output

# CLASSES OF TESTS

- **Unit testing**
  - validate each piece of program
  - **testing each function** separately
- **Regression testing**
  - add test for bugs as you find them
  - **catch reintroduced** errors that were previously fixed
- **Integration testing**
  - does **overall program** work?
  - tend to rush to do this

# TESTING APPROACHES

- **intuition** about natural boundaries to the problem

```
def is_bigger(x, y):
    """ Assumes x and y are ints
    Returns True if y is less than x, else False """
```

  - can you come up with some natural partitions?
- if no natural partitions, might do **random testing**
  - probability that code is correct increases with more tests
  - better options below
- **black box testing**
  - explore paths through specification
- **glass box testing**
  - explore paths through code

# BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0

    Returns res such that x-eps <= res*res <= x+eps """
```

- designed **without looking** at the code
- can be done by someone other than the implementer to avoid some implementer **biases**
- testing can be **reused** if implementation changes
- **paths** through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

# BLACK BOX TESTING

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

| CASE | x | eps |
|---|---|---|
| boundary | 0 | 0.0001 |
| perfect square | 25 | 0.0001 |
| less than 1 | 0.05 | 0.0001 |
| irrational square root | 2 | 0.0001 |
| extremes | 2 | 1.0/2.0**64.0 |
| extremes | 1.0/2.0**64.0 | 1.0/2.0**64.0 |
| extremes | 2.0**64.0 | 1.0/2.0**64.0 |
| extremes | 1.0/2.0**64.0 | 2.0**64.0 |
| extremes | 2.0**64.0 | 2.0**64.0 |

# GLASSBOX TESTING

- **use code** directly to guide design of test cases
- called **path-complete** if every potential path through code is tested at least once
- what are some **drawbacks** of this type of testing?
  - can go through loops arbitrarily many times
  - missing paths
- guidelines
  - branches → exercise all parts of a conditional
  - for loops → loop not entered
    body of loop executed exactly once
    body of loop executed more than once
  - while loops → same as for loops, cases that catch all ways to exit loop

# GLASSBOX TESTING

```
def abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
    if x < -1:
        return -x
    else:
        return x
```

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test boundary cases

# DEBUGGING

- steep learning curve
- goal is to have a bug-free program
- tools
  - **built in** to IDLE and Anaconda
  - **Python Tutor**
  - `print` statement
  - use your brain, be **systematic** in your hunt

# PRINT STATEMENT

- good way to **test hypothesis**
- when to print
  - enter function
  - parameters
  - function results
- use **bisection method**
  - put print halfway in code
  - decide where bug may be depending on values

# DEBUGGING STEPS

- **study** program code
  - don’t ask what is wrong
  - ask how did I get the unexpected result
  - is it part of a family?
- **scientific method**
  - study available data
  - form hypothesis
  - repeatable experiments
  - pick simplest input to testwith

# ERROR MESSAGES - EASY

- trying to access beyond the limits of a list
  `test = [1,2,3]` then `test[4]` → `IndexError`
- trying to convert an inappropriate type
  `int(test)` → `TypeError`
- referencing a non-existent variable
  `a` → `NameError`
- mixing data types without appropriate coercion
  `'3'/4` → `TypeError`
- forgetting to close parenthesis, quotation, etc.
  `a = len([1,2,3]`
  `print(a)` → `SyntaxError`