

# Python Tutorial

**CV201:** Practical session

Meitar Ronen  
Computer Science, Ben-Gurion University



*These slides are based on a  
previous version by Ron Shapira-Weber.*

# Intro to Python

Python is a general purpose programming language created nearly 30 years ago. In 2016, Python replaced Java as the most popular language in colleges and universities and since then it has never looked back.

# Intro to Python

## Why Python?

- Easy to learn
- Object Oriented approach, allows functional programming, is dynamically typed and garbage-collected
- More that 150,000 libraries are available e.g. NumPy, SciPy, OpenCV, PyBrain, SkiLearn...

# Getting Started

For compatability reasons in this class we will use the following:

- Python 3.6
- NumPy > 1.11
- SciPy > 0.18
- Matplotlib > 1.5
- OpenCV = 3.1.2

You can find installation tutorial on the course website in here.

# Getting Started

We recommend using *Anaconda* to install Python and the related packages, but it is not a must.

*Anaconda* is a free and open-source distribution of Python for scientific computing that provides simplified environment and package management and deployment. It allows users to manage virtual environments, which may contain different packages, in different versions.

If you choose to go with *Anaconda*, you can read how to manage environments and package versions [here](#), otherwise you should probably use `pip` (documentation is available [here](#)).

# Getting Started

The *Anaconda* distribution also includes the *Anaconda Navigator*, a GUI that allows users to launch applications and manage conda packages, environments and channels without using command-line commands.

The following applications are available by default in Navigator: Jupyter Lab, Jupyter Notebook, QtConsole, Spyder, Glueviz, Orange, Rstudio, Visual Studio Code.

# Getting Started

Anaconda Navigator


File Help

ANACONDA NAVIGATOR

[Sign in to Anaconda Cloud](#)

[Home](#)  
[Environments](#)  
[Learning](#)  
[Community](#)


Applications on base (root) Channels Refresh



JupyterLab  
1.1.3

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and architecture.


Launch



Jupyter  
Notebook  
6.5.1

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.


Launch



Spyder  
3.3.6

Scientific Python Development Environment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features


Launch



Glueviz  
0.13.0

Multidimensional data visualization across files. Explore relationships within and among related datasets.


Install



Orange3  
3.23.0

Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflow with a large toolbox.


Install



RStudio  
1.1.456

A set of integrated tools designed to help you be more productive with R, includes R essentials and notebooks.

Install



VS Code  
1.38.1


Streamlined code editor with support for development operations like debugging, task running and version control.

Install


# Getting Started

## Jupyter Notebook













Interactive Python Shell. It runs in the browser and combines live runnable code with narrative text equations (LaTeX), images, interactive visualizations and other rich output.

 jupyter

Labels Last Checkpoint: a day ago (autosaved)

 Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

           Code 

In [19]:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib, cv2
```

In [20]:

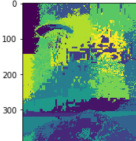
```
input_ = np.load(r'C:\Users\t-merone\Downloads\input.npy')
labels = np.load(r'C:\Users\t-merone\Downloads\labels.npy')
image = cv2.imread(r'C:\Users\t-merone\Documents\Studies\Talk\talk_bass_trax_2019\presentation_bass\Splits_and_Merges\Figures
```

In [21]:

```
labels_ = labels.reshape((481,321))
```

In [7]:

```
plt.imshow(labels_)
plt.imshow('image.jpg', labels_)
```





# Getting Started

## Jupyter Lab

Web-based user interface for Project Jupyter.

JupyterLab enables you to work with documents and activities such as Jupyter notebooks, text editors, terminals, and custom components.

The screenshot displays the JupyterLab web interface. On the left is a sidebar with a file browser showing a directory structure with files like '1024px-Hubble\_Intera...', 'bar.vi.json', 'Dockerfile', 'iris.csv', 'japan\_meteorological\_a...', 'Museums\_in\_DC.geoj...', 'README.md', and 'zika\_assembled\_geno...'. The main area is divided into two panes. The left pane shows a Jupyter notebook titled 'Data.ipynb' with a code cell containing the following Python code:

```
In [5]: 1 import pandas
2 df = pandas.read_csv('../data/iris.csv')
3 df.head(20)
```

The output of the code cell is a table with 13 rows and 6 columns:

	sepal_length	sepal_width	peta_length	peta_width	species
0	5.1	3.5	1.4	0.2	se
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
10	5.4	3.7	1.5	0.2	setosa
11	4.8	3.4	1.6	0.2	setosa
12	4.8	3.0	1.4	0.1	setosa

The right pane shows a document titled 'jupyterlab.md' with the following content:

### JupyterLab Demo

JupyterLab: The next generation user interface for Project Jupyter

<https://github.com/jupyter/jupyterlab>

It has been a collaboration between:

- Project Jupyter
- Bloomberg
- Anaconda

1) Building blocks of interactive computing

At the bottom of the interface, there are tabs for 'bar.vi.json' and '1024px-Hubble', and a large image of a galaxy.

# Getting Started

## Spyder

Open source Python IDE for scientific programming.

It includes an IPython shell (which can run multiple instances) and a variable explorer.

# Python

**And now... Some Python**



# Python: Dynamic types

Python variable assignment is different from some of the popular languages like c, c++ and java.

There is no declaration of a variable, just an assignment statement. It is possible to statically declare a variable's type, but since Python doesn't know about the type of the variable until the code is run, the declaration is of no use.

# Python: Numbers

```
1 x = 4 # 'int' is the default type.
2 # Notice there's no need for ; at the end of a statement.
3 print(x) # Prints '4'
4 print(type(x)) # Prints '<class 'int'>'
5 print(x + 1) # Addition; prints '5'
6 print(x - 1) # Subtraction; prints '3'
7 print(x * 2) # Multiplication; prints '8'
8 print(x ** 2) # Exponentiation; prints '16'
9 print(x // 2.5) # (floored) quotient of x and y; prints
    '1.0'
10 print(x % 2.5) # module/remainder of x / y; prints '1.5'
11 x += 1 # There's no x++ in Python, so this is the way to go.
    prints '5'
12 x *= 2 # prints '8'
```

# Python: Numbers

Working with floats is similar

```
1 y = 1.5
2 print(type(y)) # Prints '<class 'float'>''
3 # Casting from one type to another:
4 x = 2.5 # type(x) = 'float'
5 int(x) # Casts x to an integer; prints '2'
```

# Python: Booleans

```
1 a = True
2 b = False
3 a and b # False; equal to a & b
4 a or b # True; equal to a | b
5 not a # False
6 a != b # True
7 Print(int(a)) # prints '1'
```

# Python: Strings

```
1 hello = 'hello' # String variables can use single quotes
2 world = 'world' # or double quotes
3 print(hello) # Prints 'hello'
4 print(len(hello)) # String length; prints '5'
5 helloWorld = hello + ' ' + world # String concatenation
6 print(helloWorld) # prints 'hello world'
7 print(hello, 42) # prints 'hello 42'
8 helloWorld42 = '%s %s %d' % (hello, world, 27) # sprintf
   style string formatting
9 print(helloWorld42) # prints 'hello world 42'
10 print('{0} and {1}. Maybe even {2}.'.format('This', 'that',
   ,42)) # Print 'This and that. Maybe even 42.'
```



# Python: Data Structures

## Some more complex Data Structures



# Python: Data Structures

## Lists

Lists are used to group together items and function similar to arrays. They are capable of storing different types of items and are resizable and mutable.

```
1 squares = [1, 4, 9, 16, 25]
2 squares[0] # Python is zero-based; returns "1"
3 squares[-1] # returns the last item in the list; "25"
4 mixed_list = [4, 2.5, 'nine'] # different types of items
   could be stored in a list
5 squares + [36, 49, 64, 81, 100] # list concatenation; # "[1,
   4, 9, 16, 25, 36, 49, 64, 81, 100]"
6 squares[2] = 99 # lists are mutable; "[1, 4, 99, 16, 25]"
7
8 # A common way to add items to a list is via the append()
   method:
9 squares.append(216) # add 216 as the last value
10 squares.append(7 ** 3) # add 343 as the last value
11 # squares: [1, 8, 27, 64, 125, 216, 343]
```

# Python: Data Structures

## Lists

Slicing is an easy way to access and manipulate items in a list. Note that it returns a new (shallow) copy of the list.

```
1 squares[:] # "[1, 4, 9, 16, 25]"
2 nums = range(5) # built-in function creates a list of
  numbers; # "[0,1,2,3,4]"
3 nums_even = range(0,10,2) # "from 0 to 10 (exclusive) in
  steps of 2; "[0, 2, 4, 6, 8]"
4 even_reverse = range (10,0,-2) # "from 10 to 0 (exclusive)
  in steps of -2; "[10, 8, 6, 4, 2]"
5 nums[2:4] # Get a slice from index 2 to 4 (exclusive);"[2,
  3]"
6 nums[2:] # Get a slice from index 2 to the end; prints "[2,
  3, 4]"
7 nums[:2] # Get a slice from the start to index 2 (exclusive)
  ;"[0, 1]"
8 squares[-3:] # slicing returns a new list; "[9, 16, 25]"
9 nums[2:4] = [8, 9] # Assign a new sublist to a slice
```

# Python: Data Structures

## Lists

Lists can represent multidimensional arrays.

```
1 A = [[1,2],[3,4]] # a 2x2 array
2 A[0][1] #returns "1".
```

However, while lists can represent arrays, it doesn't mean that it should. When it comes to using arrays in Python, NumPy is the (right) way to go, and we will get to this in awhile...

# Python: Data Structures

## Loops in lists

Iterating in python feels almost like pseudo code

```
1 bag = ['notebook', 'keys', 'lipstick']
2 for stuff in bag:
3     print(stuff) #prints 'notebook', 'keys', 'lipstick'
4 # Python uses indentation to identify blocks of code
```

You can also add indices via the enumerate method

```
1 for idx, item in enumerate(bag):
2     print(idx, item) # prints '0, notebook; 1, keys; 2,
3                       lipstick'
4 for idx, _ in enumerate(bag): # _ is a throw-away variable
5     print(idx) # prints '0, 1, 2'
```

# Python: Data Structures

## Lists

While loops.

```
1 count = 0
2 while (count < 9):
3     print(count)
4     count = count + 1
```

# Python: Data Structures

## Lists

Python supports **List comprehensions** which allows creating and manipulating lists in a single line of code

```
1 S = [x**2 for x in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
2 M = [x for x in S if x % 2 == 0] # only even numbers in S ; [0, 4, 16, 36, 64]
```

# Python: Data Structures

## Dictionaries

A dictionary stores (key, value) pairs. Dictionaries are indexed by keys and not by indices, so it is best to think of a dictionary as an unordered set of (key,value) pairs.

```
1 n_seasons = {'GoT': 7, 'Friends': 10}
2 n_seasons['GoT'] # getting the value stored under the key '
    GoT'; prints '7'
3 n_seasons['Southpark'] = 'inf' # adding a new (key, value)
    item
4 print(n_seasons) # prints ' {'Friends': 10, 'GoT': 7, '
    Simpsons': 'inf'}'
5 n_seasons['GoT'] = 8 # dictionary values are mutable
6 print(n_seasons) # prints '{'Friends': 10, 'GoT': 8, '
    Simpsons': 'inf'}'
```



# Python: Data Structures

Some useful dictionary functions...

```
1 del n_seasons('Friends') # deletes the pair ('Friends', 10)
2 list(n_seasons.keys()) # returns an unsorted list of keys #
   ['Simpsons', 'GoT']
3 sorted(n_seasons.keys()) # returns a sorted list of keys #
   ['GoT', 'Simpsons']
4 'GoT' in n_seasons # True
5 'Suits' in n_seasons #False
```

# Python: Data Structures

## Loops in dictionaries

You can iterate over dictionary keys, and use list comprehensions as well.

```
1 for tv_show, seasons in n_seasons.items():
2     print(tv_show, seasons)
3 # ('Simpsons', 'inf') <-- tuple
4 # ('GoT', 8)
5 S = {x:x**2 for x in range(4)} #note the curly brackets
6 # {0: 0, 1: 1, 2: 4, 3: 9}
```

# Python: Data Structures

## Tuples

A tuple is an (immutable) ordered list of values.

```
1 t = (1,2)
2 t[0] #prints '1'
3 t = (1,2 , 'dog')
4 t[2] #prints 'dog'
5 t[2] = 'cat' # Error! tuples are immutable
```

# Python: Data Structures

## Tuples

A special problem with tuples is the construction of tuples containing 0 or 1 items. The syntax has some extra quirks to accommodate these.

Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

Ugly, but effective. For example:

```
1 empty = ()
2 singleton = 'hello', # <-- note trailing comma
3 print(len(empty)) #0
4 print(len(singleton)) #1
5 print(empty) #()
6 print(singleton) #('hello',)
```

# Python: Data Structures

## Tuples

Another example:

```
1 a = (3) # this is an int
2 b = (3,) # this is a tuple
3 print(a) # 3
4 print(type(a)) # int
5 print(b) # (3,)
6 print(type(b)) #tuple
```

## Functions

A function is created by the keyword 'def' and is followed by the function name and a list of parameters. for instance:

```
1 def powers_of_three(n):  
2     x = [] #declaring an empty list  
3     for num in n:  
4         x.append(num**3)  
5     return x  
6 numbers = range(4)  
7 # Calling the function  
8 print(powers_of_three(numbers)) #[0,1,8,27]
```

# Python:NumPy

As promised, here it comes...

# Python:NumPy

As promised, here it comes...





# Python: NumPy

## NumPy

Numpy is a core Python package which supports multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.

- For more information, visit the quick start tutorial. If you are a veteran MATLAB user, Numpy for MATLAB user is also available and is highly recommended, even for non-matlab users.

# Python: NumPy

## NumPy arrays

An Numpy array is a table of elements (usually numbers), indexed by a tuple of positive integers. In NumPy dimensions are called axes. The number of axes is the rank.

```
1 # First import numpy
2 import numpy as np #as np creates an alias
3 a = np.array([0,1,2,3]) # notice the syntax
4 b = np.arange(0,4) # similar to range, but numpy array
5 b = np.arange(0,4).astype(np.float) # creates an array of
   floats
6 print(a,b) #([0, 1, 2, 3]),[0., 1., 2., 3.])
7 a.shape #returns a tuple; (4,)
8 a.size # returns an integer; 4
9 c = np.array([[1,2,3],[4,5,6]]) # a 2x3 array, rank 2
10 c.shape # (2,3)
11 c.size # 6
```

# Python: NumPy

## Pre-defined arrays

There are number of useful pre-defined arrays that can be easily initiated using NumPy.

```
1 all_zeros = np.zeros((3,3)) # creates a 3x3 all zeros matrix
2 all_ones = np.ones((2,2)) # creates a 2x2 all ones matrix
3 all_twos = 2*np.ones((2,2)) # There's a better way to do
   this...
4 all_twos = np.full((2,2), 2) # creates a 2x2 all 2 matrix
5 identity_matrix = np.eye(3) #creates a 3x3 identity matrix
6 print(identity_matrix)
7 # ([[1., 0., 0.],
8 #  [0., 1., 0.],
9 #  [0., 0., 1.]])
```

# Python: NumPy

## Reshaping NumPy arrays

You can change the shape of an array by using the "reshape" function

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

# Python: NumPy

## Reshaping NumPy arrays

You can even ask NumPy to do the math for you

```
1 a.reshape((6,-1)) # here the -1 stands for: numpy, please do
    math for me...
2 # [[ 0 1]
3 # [ 2 3]
4 # [ 4 5]
5 # [ 6 7]
6 # [ 8 9]
7 # [10 11]]
8 b = a.reshape(2,3,-1)
9 print(b)
10 # [[[ 0 1]
11 # [ 2 3]
12 # [ 4 5]]
13 #
14 # [[ 6 7]
15 # [ 8 9]
16 # [10 11]]]
17 b.shape #(2,3,2)
```

# Python: NumPy

## Reshaping NumPy arrays

Flattening a multi-dimensional...

```
1 a = np.array([[1, 2, 3], [4, 5, 6]])
2 b = np.ravel(x)
3 print(b) #prints [1 2 3 4 5 6]
4 print(b.shape) #prints '(9L,)'
```

Or squeezing unneeded dimensions.

`np.squeeze` removes single-dimensional entries from the shape of an array.

```
1 c = np.array([[[0], [1], [2]]])
2 print(c.shape) # (1L, 3L, 1L)
3 print(c) # [[0] # [1] # [2]]
4 print(np.squeeze(c).shape) # (3L,)
5 print(np.squeeze(c)) # [0 1 2]
```

# Python: NumPy

## Indexing in NumPy

```
1 a = np.arange(10)**3
2 print(a) #array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729])
3 print(a[2]) # 8
4 print(a[2:5]) # array([ 8, 27, 64])
5 a[:6:2] = -1000 # equivalent to a[0:6:2] = -1000; from start
                  to position 6 (exclusive), set every 2nd element to
                  -1000
6 print(a) #array([-1000,  1, -1000, 27, -1000, 125, 216, 343,
                  512, 729])
7 a[ : :-1] # reversed 'a'
8 # array([ 729, 512, 343, 216, 125, -1000, 27, -1000, 1,
           -1000])
9 a = np.linspace(0,1,11) # from 0 to 1, with 11 steps
10 print(a) # [ 0.  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
11 idx = np.array([0,2,5,3])
12 print(idx) # [0 2 5 3]
13 print(a[idx]) # [ 0.  0.2  0.5  0.3]
```

# Python: NumPy

## Indexing in NumPy

Multi-dimensional arrays can have one index per axis. These indices are given in a tuple separated by commas.

```
1 # The lines below are equivalent
2 tmp = np.arange(12).reshape(3,4)
3 Print tmp
4 # array([[ 0,  1,  2,  3],
5 #        [ 4,  5,  6,  7],
6 #        [ 8,  9, 10, 11]])
7 Print(tmp[1][3])
8 print(tmp[1,3])
9 print(tmp[1][-1]) # negative indexing
10 print(tmp[1,-1]) # negative indexing
11 # prints 7
```



# Python: NumPy

## Indexing in NumPy

In NumPy you can construct an array by executing a function over each coordinate. The resulting array therefore has a value  $fn(x, y, z)$  at coordinate  $(x,y,z)$ .

```
1 # Array from function
2 def f(x,y):
3     return 10*x+y
4 b = np.fromfunction(f,(5,4),dtype=int)
5 # creates an array from a function
6 print(b)
7 # array([[ 0,  1,  2,  3],
8 #        [10, 11, 12, 13],
9 #        [20, 21, 22, 23],
10 #        [30, 31, 32, 33],
11 #        [40, 41, 42, 43]])
```

# Python: NumPy

## Indexing in NumPy

In NumPy you can construct an array by executing a function over each coordinate. The resulting array therefore has a value  $fn(x, y, z)$  at coordinate  $(x,y,z)$ .

```
1 # Array from function
2 b[2,3] # '23'
3 b[0:5, 1] # each row in the second column of b
4 #array([ 1, 11, 21, 31, 41])
5 b[ : ,1] # equivalent to the previous example
6 #array([ 1, 11, 21, 31, 41])
7 b[1:3, : ] # each column in the second and third row of b
8 #array([[10, 11, 12, 13],
9 #       [20, 21, 22, 23]])
10 #Iterating over multidimensional arrays is done with respect
    to the first axis:
11 for row in b:
12     print(row)
13 # [0 1 2 3]
14 # [10 11 12 13] etc..
```

# Python: NumPy

## Indexing in NumPy

Indexing with arrays of indices

```
1 a = np.arange(12)**2 # the first 12 square numbers
2 i = np.array( [ 1,1,3,8,5 ] ) # an array of indices
3 print(a[i]) # i can be of different shape than 'a'
4 # array([ 1, 1, 9, 64, 25])
5 # another possible syntax:
6 print(a[0], a[3]) # [0,9]
7 j = np.array( [ [ 3, 4], [ 9, 7 ] ] )
8 # a bi-dimensional array of indices
9 a[j] # the same shape as j
10 #array([[ 9, 16],
11 # [81, 49]])
```

# Python: NumPy

## Boolean indexing in NumPy

Boolean indexing can be done explicitly:

```
1 a = np.arange(5) #[0, 1, 2, 3, 4];  
2 b = np.array([0,0,1,0,1],dtype=np.bool) # needs to be the  
   same shape as 'a'  
3 # [False, False, True, False, True]  
4 print(a[b]) # array([2, 4]) # Different shape than 'a'
```

# Python: NumPy

## Boolean indexing in NumPy

Or by logical operants:

```
1 a = np.arange(12).reshape(3,4)
2 # ([[ 0, 1, 2, 3],
3 # [ 4, 5, 6, 7],
4 # [ 8, 9, 10, 11]])
5 b = a > 4
6 print(b) # b is a boolean with a's shape
7 # array([[False, False, False, False],
8 # [False, True, True, True],
9 # [True, True, True, True]], dtype=bool)
10 print(a[b]) # 1d array with the selected elements
11 array([ 5, 6, 7, 8, 9, 10, 11])
12 #This property can be very useful in assignments:
13 a[b] = 0 # All elements of 'a' higher than 4 become 0
14 print(a)
15 #array([[0, 1, 2, 3],
16 # [4, 0, 0, 0],
17 # [0, 0, 0, 0]])
```

# Python: NumPy

## Linear indexing in NumPy

Sometimes, we might want to flatten a multi-dimensional array but still use its original coordinates and vice versa. For this we can use the `unravel_index` and `ravel_multi_index` methods.

(Matlab fans will find it very similar to `sub2ind` or `ind2sub`)

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2



0,0	0,1	0,2	1,0	1,1	1,2	2,0	2,1	2,2
0	1	2	3	4	5	6	7	8

# Python: NumPy

## Linear indexing in NumPy

```
1 a_arr = np.arange(12).reshape(3,-1)
2 #array([[ 0,  1,  2,  3],
3 # [ 4,  5,  6,  7],
4 # [ 8,  9, 10, 11]])
5 a_flat = np.ravel(a_arr) #array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,
6 #  9, 10, 11])
7 idx = np.argwhere(a_flat%3==0) # returns indices for a
8 # condition
9 # print(a_flat[idx].T) # T for transpose - returns row vector
10 # array([[0, 3, 6, 9]]) # We want the indices in the dim of
11 # a_arr
12 idx_arr = np.unravel_index(idx, a_arr.shape)
13 #(array([[0], # [0], # [1], # [2]], dtype=int64),
14 # array([[0], # [3], # [2], # [1]], dtype=int64))
15 print(a_arr[idx_arr].T) #[[0 3 6 9]]
```

## Linear indexing in NumPy

```
1 # The other way around...
2 #np.ravel_multi_index Converts a tuple of index arrays into
   an array of
3 flat indices
4 idx_flat = np.ravel_multi_index(idx_arr, a_arr.shape)
5 print(idx_flat.T) # array([[0, 3, 6, 9]], dtype=int64)
```



# Python: NumPy

## Math in NumPy

We can use `max` to get the largest value, or `argmax` to the index that contains it (the same for `min` and `argmin`).

```
1 a = np.arange(6).reshape(2,3)
2 #array([[0, 1, 2],
3 # [3, 4, 5]])
4 a.max(0) # maximum element along axis 0 (columns); prints '
      array([3, 4, 5])'
5 a.max(1) # maximum element along axis 1 (rows); prints '
      array([2, 5])'
6 a.max() # maximum element of the whole array; prints 5
7 a.argmax(0) # Returns the indices of the maximum values
      along axis 0.
8 # array([1, 1, 1])
9 a.argmax(1) # array([2, 2])
10 a.argmax() # if no axis is given, the index is of the
      flattened array; prints 5
```

# Python: NumPy

## Math in NumPy

`np.maximum` is a bit different - it compares two arrays and returns a new array containing the element-wise maxima

```
1 a[0,1] = 6
2 print(a)
3 # array([[0, 6, 2],
4 #        [3, 4, 5]])
5 np.maximum(d[0,:], d[1,:]) # maximum between first and
6                             # second rows of 'a'
7 # array([3, 6, 5])
```

# Python: NumPy

## Math in NumPy

Using `np.sum` we can get the sum of an entire array, or just of a specific axis.

```
1 # a = array([[0, 6, 2],  
2 #           [3, 4, 5]])  
3 np.sum(a) # 15  
4 np.sum(a, axis = 0) # computes sum of each column; '[3, 10,  
5 #           7]'  
6 np.sum(a, axis=1) # Compute sum of each row; '[8, 12]'
```

# Python: NumPy

## Math in NumPy

`np.e` will give us the number  $e$ , while `np.exp` will generate the exp function. `np.log` is the natural log in base  $e$  (lan), and `np.log2` is log in base 2.

```
1 np.e #2.718281828459045
2 np.exp(1) #2.718281828459045
3 np.exp(np.arange(5)) # handle arrays
4 # array([1. , 2.71828183, 7.3890561 , 20.08553692,
5 #        54.59815003])
6 np.log([1, np.e, np.e**2,]) #natural log in base e = lan
7 # array([ 0., 1., 2.])
8 np.log2(8) #base 2 log # 3
```

# Python: NumPy

Beyond the basic math...

LINEAR ALGEBRA IN NUMPY

# Python: NumPy

## Linear Algebra in NumPy

Numpy has many built-in linear algebra operation which could be used on numpy arrays.

```
1 a = np.arange(1,5, dtype=float).reshape(2,2)
2 # [[ 1.  2.]
3 #   [ 3.  4.]]
4 a.T # matrix transpose
5 # [[ 1.,  3.],
6 #   [ 2.,  4.]]
7 a.transpose() # also, matrix transpose, allows for more than
8               # 2-dimensions
9 # [[ 1.,  3.],
10 #   [ 2.,  4.]]
```

# Python: NumPy

## Linear Algebra in NumPy

```
1 c = np.arange(8).reshape(2,2,-1) #shape 2x2x2
2 #array([[[0, 1],
3 # [2, 3]],
4 #
5 # [[4, 5],
6 # [6, 7]]])
7 c.transpose([2,1,0]) #order of axis to transpose
8 #array([[[0, 4],
9 # [2, 6]],
10 # [[1, 5],
11 # [3, 7]]])
```

# Python: NumPy

## Linear Algebra in NumPy

```
1 c = np.arange(8).reshape(2,2,-1) #shape 2x2x2
2 #array([[[0, 1],
3 # [2, 3]],
4 #
5 # [[4, 5],
6 # [6, 7]]])
7 c.transpose([0,2,1])
8 #array([[[0, 2],
9 # [1, 3]],
10 # [[4, 6],
11 # [5, 7]]])
```



# Python: NumPy

## Linear Algebra in NumPy

Using the `np.linalg` module we could do some more complex things, such as finding the inverse...

```
1 # a:
2 # [[ 1.  2.]
3 #   [ 3.  4.]]
4 np.linalg.inv(a) # find the matrix inverse of 'a', usually
5 computationally expensive
6 # [[-2.  ,  1. ],
7 #   [ 1.5, -0.5]])
8 b = np.full((2,2), 2)
9 a*b #element-wise multiply
10 # array([[2.,  4.],
11 #         [6.,  8.]])
```

# Python: NumPy

## Linear Algebra in NumPy

And more...

```
1 I = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
2 j = np.array([[0.0, -1.0], [1.0, 0.0]])
3 np.dot(j, j) # matrix product
4 # array([[-1., 0.],
5 # [ 0., -1.]])
6 np.trace(I) # trace # 2.0
7 np.diag(a) #vector of diagonal elements of 'a'
8 # [1., 4.]
9 v = np.array([2,3])
10 np.linalg.norm(v) # L2 norm of vector v; # 3.605551275463989
11 D,V = linalg.eig(a) # eigenvalues and eigenvectors of a
12 D,V = np.linalg.eig((a,b)) # eigenvalues and eigenvectors of
    a, b
```

# Python: NumPy

## Vector Stacking

It is possible to stack vectors on top of each other. For example, there will be cases where we will want to represent an image as a vector, instead of a matrix.

```
1 c = np.ones((1,3)) #array([[1., 1., 1.]])
2 d = 2*np.ones((1,3)) #array([[2., 2., 2.]])
3 vertical_stack = np.vstack([c,d])
4 #array([[1., 1., 1.],
5 #       [2., 2., 2.]])
6 horizontal_stack = np.hstack([c,d])
7 # array([[1., 1., 1., 2., 2., 2.]])
8 np.tile(c, (2, 3)) #create 2 by 3 copies of a
9 #array([[1., 1., 1., 1., 1., 1., 1., 1., 1.],
10 #       [1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

## PROBABILITY AND STATISTICS

# Python: NumPy

## Probability and Statistics

```
1 random_arr = np.random.random((2,2)) #creates an array with
   random values
2 random_normal = np.random.randn((2,2)) # a 2x2 sampled from
   N(0,1)
3 #output example:
4 # [[-1.25527029, 1.12880546],
5 #  [-0.78455754, -0.34960907]]
6 sigma = 2.5
7 mu = 3
8 random_normal2 = sigma*np.random.randn(2,2)+mu
9 #a 2x2 sampled from N(3,2.5)
10 # [[1.28169047, 1.64080373],
11 #  [4.76906697, 3.05345461]]
12 v = np.array([1,1,2,2,2,2,3,3,4]);
13 np.random.permutation(v) #[4, 1, 2, 3, 2, 2, 1, 2, 3]
14 np.median(a) # 2.5
15 np.mean(a) # 2.5
16 np.std(a) # 1.1180339887
17 np.var(a) # 1.25
```

# Python: NumPy

## Probability and Statistics

```
1 # Sample from an array with corresponding probabilities
   array
2 # Generate a non-uniform random sample from np.arange(5) of
   size 3:
3 np.random.choice(np.arange(5), 3, replace=False, p=[0.1, 0,
   0.3, 0.6, 0])
4 # might output array([2, 3, 0])
5 # replacing np.arange(5) with 5 yield the same result
6 np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6,
   0])
7 # might output array([2, 0, 3])
8 #
9 # replacing the replace=True allows for sampling the same
   value
10 np.random.choice(5, 3, replace=True, p=[0.1, 0, 0.3, 0.6,
   0])
11 # might output array([3, 3, 0])
```