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, SkLearn...

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.

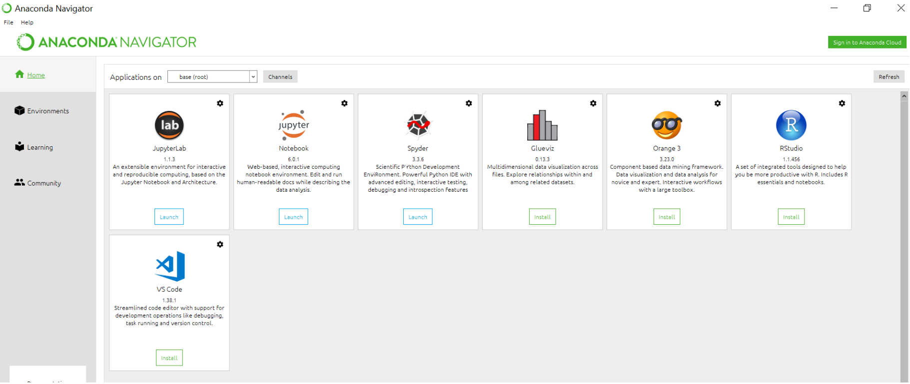
If you choose to go with *Anaconda*, you can read how to manage environments and package versions here, otherwise you should probably use pip (documentation is available here).

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.

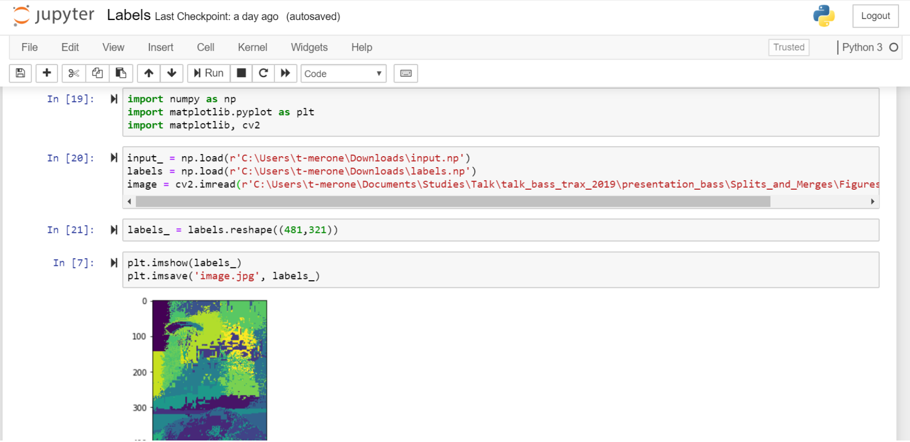
Getting Started



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.

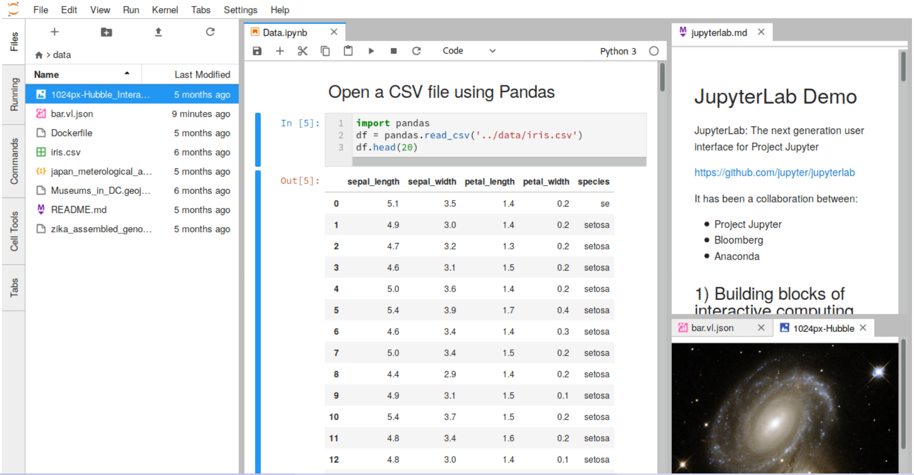


Getting Started

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.



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

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

Python: Numbers

Working with floats is similar

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

Python: Booleans

```
a = True
b = False
a and b # False; equal to a & b
a or b # True; equal to a | b
not a # False
a != b # True
Print(int(a)) # prints ‘‘1’’
```

Python: Strings

```
hello = ’hello’ # String variables can use single quotes
world = ‘‘world’’ # or double quotes
print(hello) # Prints ‘‘hello’’
print(len(hello)) # String length; prints ‘‘5’’
helloWorld = hello + ’’ + world # String concatenation
print(helloWorld) # prints ‘‘hello world’’
print(hello, 42) # prints ‘‘hello 42’’
helloWorld42 = ’%s %s %d’ % (hello, world, 27) # sprintf style string formatting
print(helloWorld42) # prints ‘‘hello world 42’’
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 resizeable and mutable.

```
squares = [1, 4, 9, 16, 25]
squares[0] # Python is zero-based; returns "1"
squares[-1] # returns the last item in the list; "25"
mixed_list = [4, 2.5, 'nine'] # different types of items could be stored in a list
squares + [36, 49, 64, 81, 100] # list concatenation; # "[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]"
squares[2] = 99 # lists are mutable; "[1, 4, 99, 16, 25]"

# A common way to add items to a list is via the append() method:
squares.append(216) # add 216 as the last value
squares.append(7 ** 3) # add 343 as the last value
# 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.

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

Python: Data Structures

Lists

Lists can represent multidimensional arrays.

```
A = [[1,2],[3,4]] # a 2x2 array
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

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

You can also add indices via the enumerate method

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

Python: Data Structures

Lists

While loops.

```python
count = 0
while (count < 9):
  print(count)
  count = count + 1
```

Python: Data Structures

Lists

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

```
S = [x**2 for x in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49,
    64, 81]
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.

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

Python: Data Structures

Some useful dictionary functions...

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

Loops in dictionaries

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

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

Python: Data Structures

Tuples

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

```
t = (1,2)
t[0] #prints '1'
t = (1,2 ,'dog')
t[2] #prints 'dog'
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:

```
empty = ()
singleton = 'hello', # <-- note trailing comma
print(len(empty)) #0
print(len(singleton)) #1
print(empty) #()
print(singleton) #(`hello',)
```

Python: Data Structures

Tuples

Another example:

```python
a = (3) # this is an int
b = (3,) # this is a tuple
print(a) # 3
print(type(a)) # int
print(b) # (3,)
print(type(b)) #tuple
```

Python: Data Structures

Functions

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

```
def powers_of_three(n):
  x = [] #declaring an empty list
  for num in n:
    x.append(num**3)
  return x
numbers = range(4)
# Calling the function
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.

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

Python: NumPy

Pre-defined arrays

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

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

Python: NumPy

Reshaping NumPy arrays

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

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

Python: NumPy

Reshaping NumPy arrays

You can even ask NumPy to do the math for you

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

Python: NumPy

Reshaping NumPy arrays

Flattening a multi-dimensional...

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

Or squeezing unneeded dimensions.

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

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

Python: NumPy

Indexing in NumPy

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

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

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

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

Python: NumPy

Indexing in NumPy

Indexing with arrays of indices

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

Python: NumPy

Boolean indexing in NumPy

Boolean indexing can be done explicitly:

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

Python: NumPy

Boolean indexing in NumPy

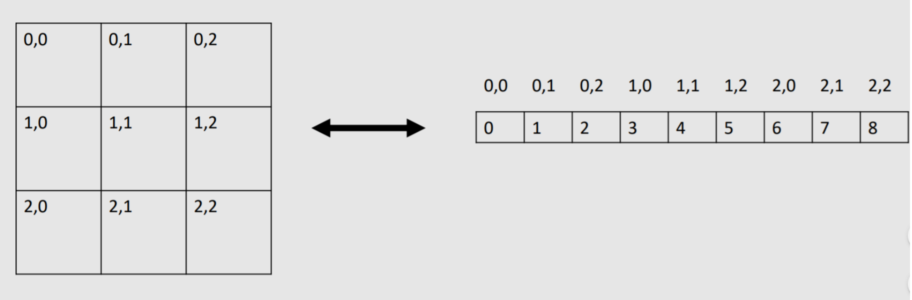
Or by logical operants:

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



Python: NumPy

Linear indexing in NumPy

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

Python: NumPy

Linear indexing in NumPy

```
# The other way around...
#np.ravel_multi_index Converts a tuple of index arrays into an array of
flat indices
idx_flat = np.ravel_multi_index(idx_arr, a_arr.shape)
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`).

```
a = np.arange(6).reshape(2,3)
#array([[0, 1, 2],
# [3, 4, 5]])
a.max(0) # maximum element along axis 0 (columns); prints 'array([3, 4, 5])'
a.max(1) # maximum element along axis 1 (rows); prints 'array([2, 5])'
a.max() # maximum element of the whole array; prints 5
a.argmax(0) # Returns the indices of the maximum values along axis 0.
# array([1, 1, 1])
a.argmax(1) # array([2, 2])
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

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

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

```
np.e #2.718281828459045
np.exp(1) #2.718281828459045
np.exp(np.arange(5)) # handle arrays
# array([1. , 2.71828183, 7.3890561 , 20.08553692,
    54.59815003])
np.log([1, np.e, np.e**2,]) #natural log in base e = lan
# array([ 0., 1., 2.])
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 bulit-in linear algebra operation which could be used on numpy arrays.

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

Python: NumPy

Linear Algebra in NumPy

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

Python: NumPy

Linear Algebra in NumPy

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

Linear Algebra in NumPy

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

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

Python: NumPy

Linear Algebra in NumPy

And more...

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

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

Python: NumPy

Probability and Statistics

Python: NumPy

Probability and Statistics

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

Python: NumPy

Probability and Statistics

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