

CS224N

Python Introduction

Plan for Today

- Intro to Python
- Installing Python
- Python Syntax
- Numpy
- Python Demo

Intro to Python

What is Python?

- General-purpose, high-level scripting language
- Used for a wide variety of purposes including networking and web applications
- Most popular in the scientific community for its ease of use

Pros vs Cons

- Pros:

- Easy to understand and write, very similar to English
- Works across systems (Windows, Mac, Linux)
- Object Oriented
- Really great standard library
- Dynamically typed?

- Cons:

- Python can be slow
- Not great for mobile development
- Dynamically typed?

Installing Python

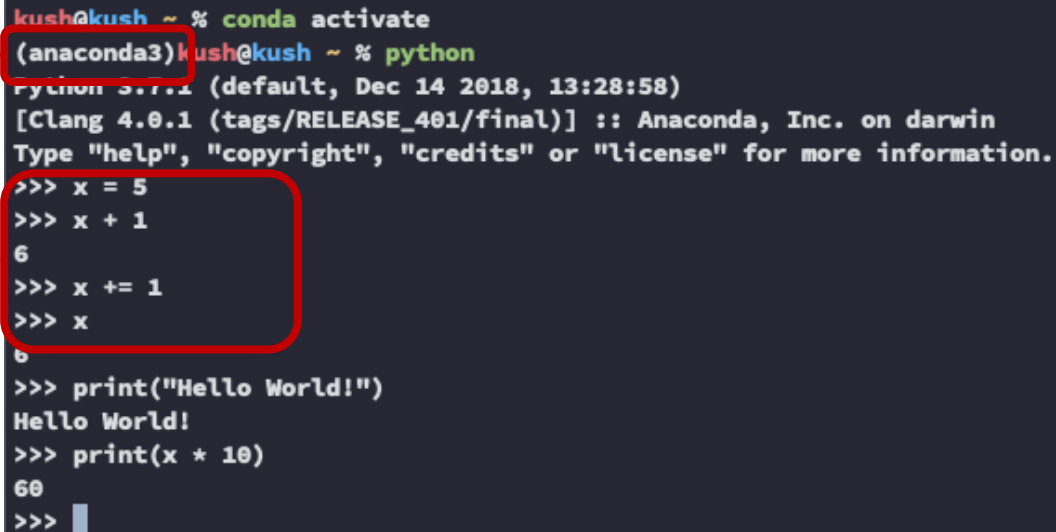
Installing and Running Python

- Download from Anaconda:
<https://www.anaconda.com/distribution/>
 - Includes Python, as well as several packages for scientific computing
- In your terminal, start up the Anaconda installation of Python:
 - `conda activate`
- Because Python is a scripting language, you can try it out right in the terminal; just type: `python`
- Follow instructions on Assign1 to create 'environments'
 - Help keep your projects separated so there aren't conflicting installations!

Check Your Installation

Which
environment I
am using (this is
the default)

Python in the
terminal! This will
be helpful for
Numpy when you
want to test
broadcasting
(more on this
later)



```
kush@kush ~ % conda activate  
(anaconda3)kush@kush ~ % python  
Python 3.7.1 (default, Dec 14 2018, 13:28:58)  
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> x = 5  
>>> x + 1  
6  
>>> x += 1  
>>> x  
6  
>>> print("Hello World!")  
Hello World!  
>>> print(x * 10)  
60  
>>> 
```

The terminal output shows the process of activating the conda environment and running a python script. Two red annotations highlight specific parts: a red box around the prompt `(anaconda3)` with an arrow pointing to the text "Which environment I am using (this is the default)", and another red box around the code block `>>> x = 5` through `>>> x` with an arrow pointing to the text "Python in the terminal! This will be helpful for Numpy when you want to test broadcasting (more on this later)".

Writing Programs

- For longer tasks, probably want to write in a program that you can run on command
- To write programs, people often use IDEs
 - Pycharm (can get professional version for free since you are a student!)
 - Sublime (after modification with plugins)
 - VSCode (after modification with plugins)
- IDEs include lots of nice bells and whistles like code completion, syntax checking and a debugger
- If you choose to just use a text editor, you can run your program from the terminal by using the command:
`python <filename>.py`

Basic Python

Basic data structures

```
example_list = [1, 2, '3', 'four']  
example_set = set([1, 2, '3', 'four', 'four'])  
example_dictionary = {  
    '1': 'one',  
    '2': 'two',  
    '3': 'three'  
}
```

- None of these types have a fixed type: can contain anything
- Sets will remove duplicates; only one copy of 'four'

More on Lists

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
three = list_of_lists[0][2]
four = list_of_lists[1][0]

my_list = [i for i in range(10)]
my_list2 = [i**2 for i in range(10)]
initialize_2d_list = [[i + j for i in range(5)] for j in range(10)]
```

- Can easily create 2D arrays and then index into them
- List comprehensions are a slick way to create lists

Sorting Lists

```
random_list = [3, 12, 5, 6]
sorted_list = sorted(random_list)

random_list = [(3, 'A'), (12, 'D'), (5, 'M'), (6, 'B')]
sorted_list = sorted(random_list, key=lambda x: x[1])
```

- Sorted function lets you sort a list
- Has additional 'key' parameter to which you can pass a function that tells sorted how to compare
- For more details, look up 'lambda functions'

Functions, Loops and Control Flow

Integers in $[0, a)$

Boolean
statements

Integers from 1
(inclusive) to b
(exclusive),
counting by 2

```
def myFunction(a, b):  
    for num1 in range(a):  
        if num1 % 2 == 0 and num1 % 4 == 0:  
            print(str(num1) + " is multiple of four!")  
        elif num1 % 2 == 0 and num1 % 4 != 0:  
            print(str(num1) + " is even, but not a multiple of four!")  
        else:  
            print(str(num1) + " is odd!")  
    print("-" * 10)  
    for num2 in range(1, b, 2):  
        print(num2, 2**num2)  
  
def main():  
    a = 5  
    b = 10  
    myFunction(a, b)  
  
if __name__ == '__main__':  
    main()
```

If called from command line

Classes

Initialize the class to
get an **instance** using
some parameters

Instance variable

Does something
with the **instance**

```
class Vehicle:
    def __init__(self, make, name, year,
                 is_electric=False, price=100):
        self.name = name
        self.make = make
        self.year = year
        self.is_electric = is_electric
        self.price = price

        self.odometer = 0

    def drive(self, distance):
        self.odometer += distance

    def compute_price(self):
        if self.is_electric:
            price = self.price / (self.odometer * 0.8)
        else:
            price = self.price / self.odometer
        return price
```

To use a class

Instantiate a class,
get an **instance**

```
family_car = Vehicle('Honda', 'Accord', '2019',  
                    price=10000)
```

```
print(family_car.compute_price())  
family_car.drive(100)  
print(family_car.compute_price())
```

Call an instance method

Numpy & Scipy

What is Numpy? What is Scipy?

- Numpy – package for vector and matrix multiplication
- Scipy – package for scientific and technical computing
- The main advantage of numpy and scipy are their speed
- Speed comes from efficient memory representation and low-level machine instructions

Ndarray

- Most important structure in numpy
- Can only contain one type of value
- Extremely fast
- Used to represent vectors, matrices, tensors
- Calling `myArray.shape` will return a tuple of integers that represent the shape of the ndarray – very important!

```
ones = np.ones(10)
randomMatrix = np.random.rand(5, 10)
fromPythonList = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

Ndarray – Addition (same shape)

- When two Ndarrays are the same shape, addition acts exactly as you would expect: component wise!
- We will discuss the case when they are not the same shape later

Ndarray – Addition (same shape, example 1)

```
>>> x = np.array([1, 0, 0, 1])
>>> y = np.array([-1, 5, 10, -1])
>>> x
array([1, 0, 0, 1])
>>> y
array([-1,  5, 10, -1])
>>> x + y
array([ 0,  5, 10,  0])
```

Ndarray – Addition (same shape, example 2)

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

Ndarray – Component-Wise Multiplication (same shape)

- When two ndarrays are the same dimension and you use the python multiplication operator (*) you get component-wise multiplication, not matrix multiplication!
- When we get to neural networks, we will see that this Hadamard product is very important

Ndarray – Component-Wise Multiplication (same shape, example 1)

```
>>> A = np.array([[5, 10], [3, 4]])
>>> B = np.array([[6, 20], [-4, -5]])
>>> A
array([[ 5, 10],
       [ 3,  4]])
>>> B
array([[ 6, 20],
       [-4, -5]])
>>> A * B
array([[ 30, 200],
       [-12, -20]])
```


Ndarray – np.dot

- Vector-Vector, Matrix-Vector and Matrix-Matrix products are calculated using `np.dot()`
- As with most numpy functions, they behave differently depending on the shapes of the input; we will look at the most common uses

Ndarray – np.dot with two vectors

- When the two inputs to `np.dot()` are both 1d vectors, then the result is the standard dot product

```
>>> x = np.array([1, 2, 3, 4])
>>> y = np.array([5, 10, 15, 20])
>>> np.dot(x, y)
150
>>> sum([i * j for (i, j) in zip(x, y)])
150
```

Ndarray – np.dot with matrix and vector

- In this case, `np.dot()` acts as matrix-vector multiplication
- Note that dimensions matter!

```
>>> A = np.array([[1, 10], [2, 5], [3, 3]])
>>> A
array([[ 1, 10],
       [ 2,  5],
       [ 3,  3]])
>>> x
array([3, 4])
>>> np.dot(A, x)
array([43, 26, 21])
```

Ndarray – np.dot with two matrices

- Here, we have standard matrix multiplication
- However, numpy documentation says that it is preferable to use `np.matmul()`

Ndarray – np.dot with two matrices (example)

```
>>> A = np.array([[1, 5], [2, 3], [3, 10]])
>>> B = np.array([[3, 4], [4, 5]])
>>> A
array([[ 1,  5],
       [ 2,  3],
       [ 3, 10]])
>>> B
array([[3, 4],
       [4, 5]])
>>> np.dot(A, B)
array([[23, 29],
       [18, 23],
       [49, 62]])
>>> np.matmul(A, B)
array([[23, 29],
       [18, 23],
       [49, 62]])
```

Broadcasting

- In math, operations like dot products and matrix addition require the same dimensions. In numpy, this is not the case
- Up until now, we have used 1d and 2d ndarrays, representing vectors and matrices, and numpy acts as we would expect
- However, the operations we have described work even when the two inputs do not have 'standard' shapes, given by a set of very specific rules

General Broadcasting Rules

- Write out the shapes of each ndarray
- Starting from the back, that dimension has compatible values if either:
 - They are the same value, or
 - One of them is a 1
- The size of the resulting array is the maximum along each dimension
- Note: the two ndarrays do not need to have the same number of dimensions

Broadcasting – Example 1 (easiest)

- In this case, we add a scalar to an ndarray
- Numpy automatically adds the scalar to every single element

```
>>> x = np.array([1, 10, 15, 100])  
>>> x + 10  
array([ 11,  20,  25, 110])
```


Broadcasting – Example 2 (medium)

```
>>> A = np.array([[1, 10], [15, 20], [25, 50]])
>>> x = np.array([5, 100])
>>> A.shape
(3, 2)
>>> x.shape
(2,)
>>> A
array([[ 1, 10],
       [15, 20],
       [25, 50]])
>>> x
array([ 5, 100])
>>> A + x
array([[ 6, 110],
       [20, 120],
       [30, 150]])
```

Broadcasting – Example 3 (hardest)

- From the `np.matmul()` documentation:
 - If either argument is N-D, $N > 2$, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- What will be the dimension of the output for a call with the following shapes?
 - (1, 5, 6), (6, 7)
 - (3, 5, 6), (6, 7)
 - (3, 5, 6), (3, 6, 7)
 - (3, 4, 5, 6), (6, 7)
 - (3, 4, 5, 6), (4, 6, 7)
 - (3, 4, 5, 6), (1, 4, 6, 7)

Broadcasting – Example 3 (hardest, one answer)

- Take the fifth example, the shapes are (3, 4, 5, 6) and (4, 6, 7)
- According to the documentation, the last two dimensions represent matrices, so we take those out and broadcast the rest: (3, 4) and (4,)
- Using our broadcasting rules, the result of broadcasting these shapes will be (3, 4)
- Matrix multiplication results in a matrix of shape (5, 7)
- Our output will have shape (3, 4, 5, 7)

Mathematical Functions on Ndarrays

- Numpy has a wide array of mathematical functions that you can apply to arrays

```
>>> x = np.array([0, np.pi / 4, np.pi / 2, np.pi, 3 * np.pi / 2, 2 * np.pi])
>>> x
array([0.          , 0.78539816, 1.57079633, 3.14159265, 4.71238898,
       6.28318531])
>>> np.sin(x)
array([ 0.00000000e+00,  7.07106781e-01,  1.00000000e+00,  1.22464680e-16,
       -1.00000000e+00, -2.44929360e-16])
```

Mathematical Functions on Ndarrays – cont.

- Some functions, like `sum` and `max`, can be applied along a given axis
- Applying along that dimension gets rid of that dimension, and replaces it with the function applied across that dimension

```
>>> A = np.array([[1, 3], [2, 4], [3, 5]])
>>> A
array([[1, 3],
       [2, 4],
       [3, 5]])
>>> np.sum(A, axis=0)
array([ 6, 12])
>>> np.sum(A, axis=1)
array([4, 6, 8])
```

Numpy Speed – Dot Product

```
a = np.array([i for i in range(10000)])
b = np.array([i for i in range(10000)])

tic = time.time()
dot = 0.0
for i in range(len(a)):
    dot += a[i] * b[i]
toc = time.time()

print("dot_product = "+str(dot));
print("Computation time = " + str(1000*(toc - tic )) + "ms")

n_tic = time.time()
n_dot_product = np.array(a).dot(np.array(b))
n_toc = time.time()

print("\nn_dot_product = "+str(n_dot_product))
print("Computation time = "+str(1000*(n_toc - n_tic ))+"ms")
```

```
dot_product = 333283335000.0
Computation time = 5.955934524536133ms

n_dot_product = 333283335000
Computation time = 0.0591278076171875ms
```

Numpy Speed – Applying a Function

```
myListFor = [i for i in range(100000)]
tic = time.time()
for i in range(len(myListFor)):
    myListFor[i] = np.sin(myListFor[i])
toc = time.time()

myListMap = [i for i in range(100000)]
mtic = time.time()
myListMap = list(map(np.sin, myListMap))
mtoc = time.time()

myListNumpy = [i for i in range(100000)]
numpytic = time.time()
myListNumpy = np.sin(myListNumpy)
numpytoc = time.time()

print("for_loop = " + str(1000*(toc - tic)) + "ms")
print("map = " + str(1000*(mtoc - mtic)) + "ms")
print("numpy = " + str(1000*(numpytoc - numpytic)) + "ms")
```

```
for_loop = 107.09214210510254ms
map = 83.14704895019531ms
numpy = 7.506370544433594ms
```

Popular usage, read before use!

Python Command	Description
<code>scipy.linalg.inv</code>	Inverse of matrix (numpy as equivalent)
<code>scipy.linalg.eig</code>	Get eigen value (Read documentation on eig and numpy equivalent)
<code>scipy.spatial.distance</code>	Compute pairwise distance
<code>np.matmul</code>	Matrix multiply
<code>np.zeros</code>	Create a matrix filled with zeros (Read on <code>np.ones</code>)
<code>np.arange</code>	Start, stop, step size (Read on <code>np.linspace</code>)
<code>np.identity</code>	Create an identity matrix
<code>np.vstack</code>	Vertically stack 2 arrays (Read on <code>np.hstack</code>)

Your friend for debugging

Python Command	Description
<code>array.shape</code>	Get shape of numpy array
<code>array.dtype</code>	Check data type of array (for precision, for weird behavior)
<code>type(stuff)</code>	Get type of a variable
<code>import pdb; pdb.set_trace()</code>	Set a breakpoint (https://docs.python.org/3/library/pdb.html)
<code>print(f'My name is {name}')</code>	Easy way to construct a message

Advice

- If you are unsure how an operation will work on ndarrays of a certain shape, try it out!
- Create random matrices that have the shape you are looking at, do the operation, and check the shape of the output
- Python scripting in the terminal is great for this!

Plotting

More Tools

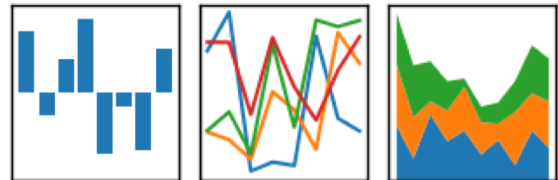
- Scatter plot
- Line plot
- Bar plot (Histogram)
- 3D plot

matplotlib



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

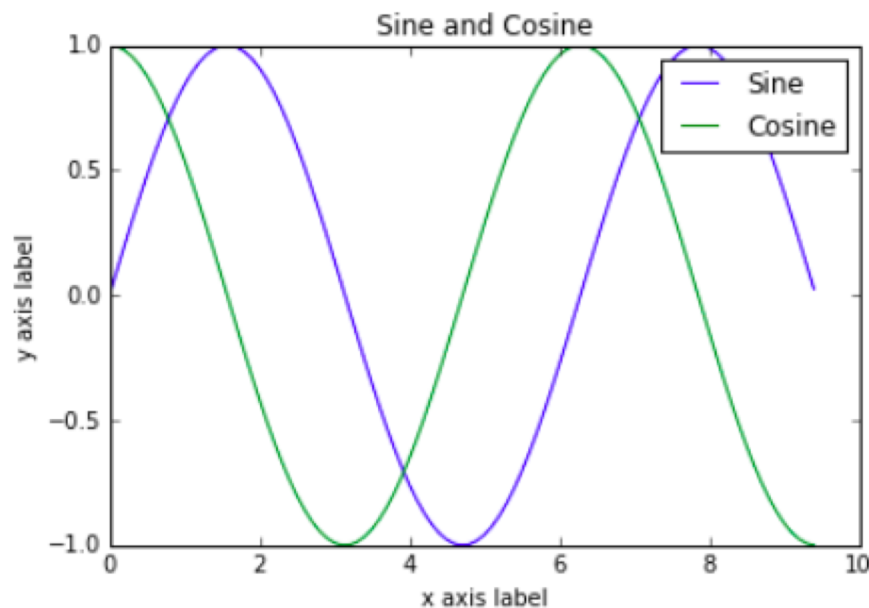


Plotting Functions

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

# Compute the x and y coordinates for po
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



Python Live Demo in PyCharm

- Calculate the eigenvector associated with the dominant eigenvalue
- Use the power iteration method:
- $$b_{k+1} = \frac{Ab_k}{||Ab_k||}$$
- (If not at live session, can download the code from course website)

Links

[Python Documentation](#)

[Numpy Reference](#)

[CS 231N Python Tutorial](#)

[Download Pycharm](#)