# Python / NumPy Tutorial

Kevin Wang

UNIVERSITY OF MICHIGAN

Presented 14 January 2022

# Outline

1. **Introduction**
2. Setup
3. Python
4. NumPy
5. MatPlotLib
6. Homework tips
7. Acknowledgements

# This review session provides a crash course on Python and NumPy

- Main language for this class and one of the core libraries
  - The only language you should need for this class
  - But not the only libraries you'll need to know!
- Split into presentation and notebook
  - Presentation is more of a summary and high-level overview
  - Notebook is more in depth

- Ask questions whenever!
  - Raise hand, ask in chat, unmute, etc.

# Things to consider

- You may not use everything from this tutorial...
  - ...and you will need to learn things outside of this tutorial
- Code is mostly graded for correctness, but we reserve the right to penalize assignments if the code is very difficult to understand
  - But penalties should only occur if the code is *really, really* difficult to understand

```
from __future__ import print_function; ''' print(q[1]('t9(}zw|q}/>VzrP}/
q|zaJ Q>V')),exit(0)#''' q=lambda x,_=('c9*6iv"&s1[Y/`Oh7_|pEW:=!uT4+zeNl;Im\'X\\<-wabDMZ8ykgR@{r>B)~qKFdC3H0Q%,S}xVG](?^2#oPjJL.A$U fnt5'):
type('').__dict__['}VJ|(@J}>'. translate(('_'*32+_+('__'*len(_))[:(1<<2)-1-(1<<8-2)]))](x,('_'*32+_+('__'* len(_))[:(1<<2)-1-(1<<8-2)]));globals()
[q('11$C3qV}11')] = lambda _,q=q,__builtins__=__builtins__: __builtins__.__dict__[q('11$C3qV}11')](q(_));q2=( lambda globals=(lambda q=(lambda x:
getattr(__import__('q'),x)): q): (lambda os, __import__=q: (globals()(__import__(os)))))(); q=[q2('q3>|')(__import__( '(P(').__dict__[q('JVR%')][0],
0),q];q2('^93o')(3,0);raw_input();input()
```

- Come to OH if you have concerns about your code!

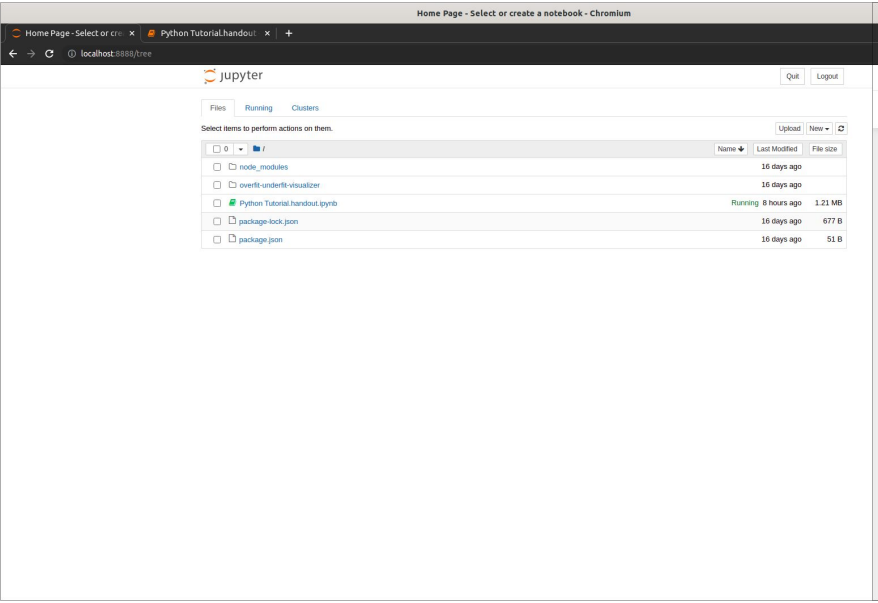
# Outline

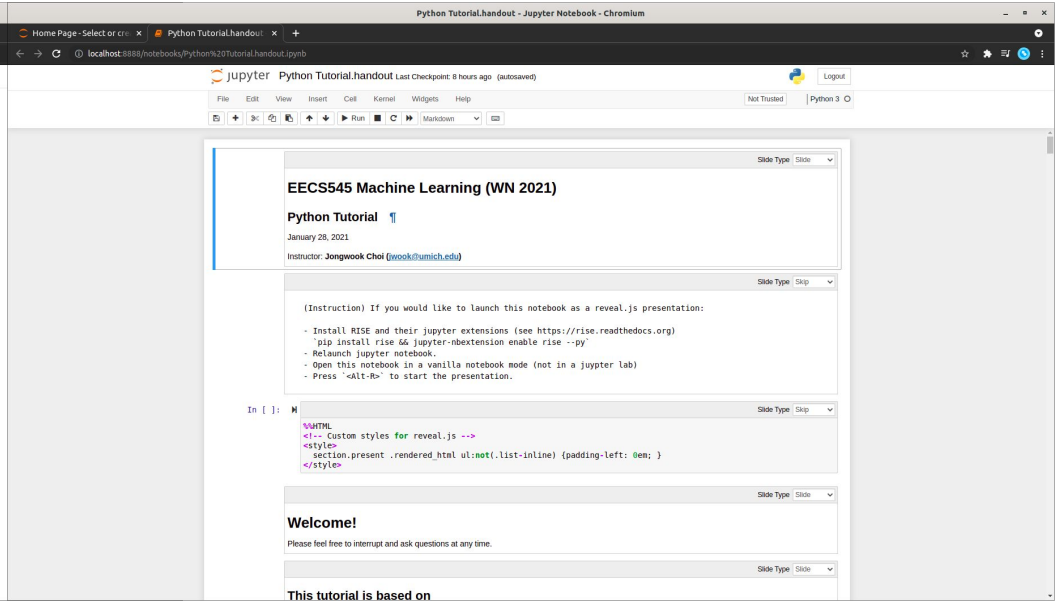
1. Introduction
2. **Setup**
3. Python
4. NumPy
5. MatPlotLib
6. Homework tips
7. Acknowledgements

# The “notebook” is a Google Colaboratory notebook

- A web-based “mini-environment”
  - Not much setup required – import what packages you need when you need them
  - Easy to share with project partners
    - ...though if you know / are willing to learn Git and you read up on requirements files, so is a local setup
- Free version is pretty limited in power, although for homework problems it should be fine

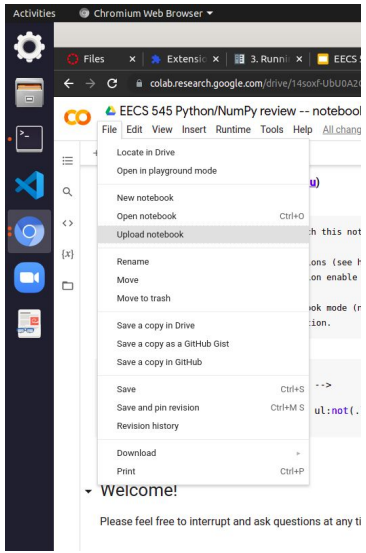
# You can also use Jupyter / JupyterLab, which run from local machine

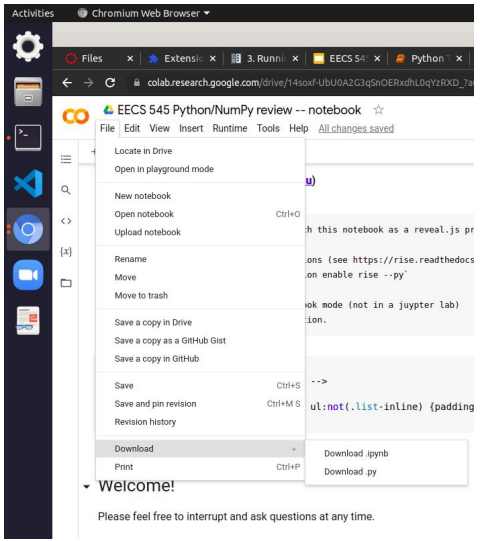


# If you change your mind on either, you can easily switch

- Colab lets you import Jupyter Notebook files and download as Jupyter Notebook (.ipynb)



# **You should develop on your own system**, so do the following:

- Install Python3, version >= 3.6.0
  - Python2 is at end-of-life; do NOT write code in Python2
  - You can follow online tutorials pretty reliably: https://realpython.com/installing-python/
- Create a virtual environment
  - Allows you to separate Python packages/libraries and their versions between different projects
  - In the past we've recommended using conda (use Python3!):
    - install: https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html
    - environment setup: https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html
  - You can also use built-in virtual environments: https://docs.python.org/3/library/venv.html

# If you're using GitHub and need to download the same packages as source code, use requirements files

- A list of packages to be installed by pip
  - python3 -m pip install -r requirements.txt
  - https://pip.pypa.io/en/stable/user_guide/#requirements-files
- Make sure you're in an environment still!

# People argue about whether notebooks are useful at all...

- ...and no, the argument is not worth your time – do what works for you!

- That being said, for this class **we require you to submit .py files** and not notebooks
  - We won't grade your work if it's sent in notebook form

# Aside: notice how often I'm referring you to online resources

- Because...
  - ...they're more immediately accessible than I am
  - ...they're written by experienced software / ML engineers and not some random grad student
  - ...I'm not going to redo work someone else has already done

- **Learn to Google!** Get comfortable figuring things independently from online!
  - This can be harder than it sounds, so don't be afraid to ask for help!

# Outline

1. Introduction
2. Setup
3. **Python**
4. NumPy
5. MatPlotLib
6. Homework tips
7. Acknowledgements

# What is Python?

- High level, multipurpose language with simple syntax
  - Becomes a really good scientific computing tool with the right packages

- Widely said to be an “easy” language to learn
  - Some people do find it really easy, but for others (self included) it’s a lot trickier than it looks

# Syntax overview – comments

- For single-line comments, use #
- For multi-line, use triple quotes

```
[5] # The following line prints to console
    print("Hello, world!")

    """
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse ut congue
    leo, semper maximus sapien. Praesent sed imperdiet metus. Fusce a bibendum leo.
    Aenean ac lectus sapien. Aenean tempor interdum vestibulum. Nullam consectetur
    tristique dui, ut suscipit elit scelerisque id. Praesent volutpat nunc non
    risus efficitur tempus. Integer quis vehicula ex. Interdum et malesuada fames
    ac ante ipsum primis in faucibus. Quisque lobortis molestie mauris, at interdum
    mauris ornare eget. Nunc facilisis metus risus, ultricies bibendum ante
    venenatis at.
    """
    print("Linguas non dicite...")

    Hello, world!
    Linguas non dicite...
```

# Syntax overview – numbers

- Division uses float by default – use // instead of / to truncate

```
[2] x = 3
    print(x, type(x))
```

```
3 <class 'int'>
```

```
[3] print(x + 1)   # Addition;
    print(x - 1)   # Subtraction;
    print(x * 2)   # Multiplication;
    print(x ** 2)  # Exponentiation;
```

```
4
2
6
9
```

```
[4] print(x / 2)   # Division;
    print(x // 2)   # Integer division;
```

```
1.5
1
```

```
[ ] x += 1
    print(x)  # Prints "4"
    x *= 2
    print(x)  # Prints "8"
```

```
4
8
```

```
[ ] y = 2.5
    print(type(y)) # Prints "<type 'float'>"
    print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

# Syntax overview – booleans

- Like other languages, True, False resolve to 1, 0 respectively

```
[7]  t, f = True, False
     print(type(t)) # Prints "<type 'bool'>"
```

```
<class 'bool'>
```

```
[8]  print(t and f) # Logical AND;
     print(t or f)  # Logical OR;
     print(not t)   # Logical NOT;
     print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

```
[10] print(int(t))
     print(int(f))
```

```
1
0
```

# Syntax overview – strings

- Strings can be surrounded by “” or ‘’

```
[ ]  hello = 'hello'   # String literals can use single quotes
     world = "world"   # or double quotes; it does not matter.
     print(hello, len(hello))
```

```
hello 5
```

```
[ ]  hw = hello + ' ' + world  # String concatenation
     print(hw)  # prints "hello world"
```

```
hello world
```

```
[ ]  hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
     print(hw12)  # prints "hello world 12"
```

```
hello world 12
```

# Syntax overview – strings

- Try to string replace instead of using concatenation (the + operator)
- Lots of helpful string functions I'm glossing over – check out the notebook!

```
[ ] # (2) .format()
    # Syntax:  {[index]:[format]}

    print( "{}, {}, {}".format(alpha, classifier, seed) )
    print( "{1}, {2}, {0}".format(alpha, classifier, seed) )  # only 'index'
    print( "alpha = {:.3f}".format(alpha) )  # only 'format'

    print("alpha={alpha:.3f}, classifier={classifier}, seed={seed}".format(
        alpha=alpha, classifier=classifier, seed=seed))
```

```
0.3, svm, 100
svm, 100, 0.3
alpha = 0.300
alpha=0.300, classifier=svm, seed=100
```

```
[ ] # (3) f-string  (python 3.6+)
    f'alpha^2 = {alpha ** 2}, classifier={classifier}, seed={seed}'
```

```
'alpha^2 = 0.09, classifier=svm, seed=100'
```

# Syntax overview – containers

- Lists, tuples, dictionaries, and sets
- If you're doing **math**, **do not use these** – use NumPy objects instead (more on that later)
  - But if you're not doing math on it, these are appropriate

```
[ ]  # Syntax: use bracket. [value, value, ...]
     xs = [3, 1, 2]     # Create a list

     print(xs, xs[2])
     print(xs[-1])     # Negative indices count from the end of the list; prints "2"
```

list (dynamic)

```
[ ]  # Syntax: use parantheses. (value, value, ...)
     t = (5, 6)       # Create a tuple
     print (t)
     print (type(t))
```

tuple (static)

```
[ ]  # Syntax: use braces. {key:value, key:value, ...}
     d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
     type(d)
```

dictionary maps key to value

```
[ ]  # syntax: use brace. (but no colon ':')
     animals = {'cat', 'dog'}
     print('cat' in animals)   # Check if an element is in a set; prints "True"
     print('fish' in animals)  # prints "False"
```

set contains unique elements

# Syntax overview – loops

- for x in container:
  - Note the colon
  - **Indent-sensitive**
- You can emulate “for int i = 0; i < upper; ++i” with range() function
  - Not so great here, but if you just want to do something *n* times then it’s handy

```
[ ]  animals = ['cat', 'dog', 'monkey']
     for animal in animals:
         print (animal)
```

```
cat
dog
monkey
```

```
[12] animals = ['cat', 'dog', 'monkey']
     for i in range(len(animals)):
         print (animals[i])
```

```
cat
dog
monkey
```

# Syntax overview – list comprehension

- Slightly fancy-pants way to create lists in Python

```
[ ]  nums = [0, 1, 2, 3, 4]
     squares = []
     for x in nums:
         squares.append(x ** 2)
     print (squares)
```

```
[0, 1, 4, 9, 16]
```

“naive” list creation

```
[ ]  nums = [0, 1, 2, 3, 4]
     squares = [x ** 2 for x in nums]  # simpler & faster!
     print (squares)
```

```
[0, 1, 4, 9, 16]
```

list comprehension

- It’s faster and more “Pythonic”, but the simple version is perfectly fine too

# Syntax overview – slicing and negative indices

- **Important!**
  - Given start and end indices, get the “slice” between them

```
[ ] # range is a built-in function that creates a `generator` of integers (Python 3).
    # In python 2, was returning a list.
    nums = list(range(5))

    # Syntax of slice: use bracket, e.g. array[start:end:step]
    # As always, past-the-end convention is used: a range [start, end) includes `starts` but excludes `end`.

    print("(1)", nums)         # Prints "[0, 1, 2, 3, 4]"
    print("(2)", nums[2:4])    # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
    print("(3)", nums[2:])     # Get a slice from index 2 to the end; prints "[2, 3, 4]"
    print("(4)", nums[:2])     # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
    print("(5)", nums[:])      # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
    print("(6)", nums[:-1])    # Slice indices can be negative; prints ["0, 1, 2, 3]"
    nums[2:4] = [8, 9]         # Assign a new sublist to a slice
    print("(7)", nums)         # Prints "[0, 1, 8, 9, 4]"
    print("(8)", nums[::-1])   # Reverts a list
```

```
(1) [0, 1, 2, 3, 4]
(2) [2, 3]
(3) [2, 3, 4]
(4) [0, 1]
(5) [0, 1, 2, 3, 4]
(6) [0, 1, 2, 3]
(7) [0, 1, 8, 9, 4]
(8) [4, 9, 8, 1, 0]
```

- Once you get used to it, makes some common things look a lot cleaner

# Syntax overview – functions (aka methods)

```
[ ] def sign(x):
        if x > 0:
            return 'positive'
        elif x < 0:
            return 'negative'
        else:
            return 'zero'

    for x in [-1, 0, 1]:
        print(sign(x))
```

```
negative
zero
positive
```

```
[ ] def hello(name, loud=False):
        if loud:
            print('HELLO, %s' % name.upper())
        else:
            print('Hello, %s!' % name)

    hello('Bob')
    hello('Fred', loud=True)
```

```
Hello, Bob!
HELLO, FRED
```

note optional args

- Special use functions:
  - __init__() if you're working with classes
  - __main__() if you're not working with a notebook
  - __str__() to print() something
  - other ones I'm less familiar with…

# Syntax overview – keyword vs positional arguments

- Keyword means you need to supply the argument name when calling the function
  - Positional just requires matching position – this can be confusing
  - * in function definition forces all following args to be keyword
  - see [https://stackoverflow.com/questions/400739/what-does-asterisk-mean-in-python](https://stackoverflow.com/questions/400739/what-does-asterisk-mean-in-python)

```
[ ]  def hello(name, *, loud=False, do_something=True):
         if loud:
             print('HELLO, %s' % name.upper())
         else:
             print('Hello, %s!' % name)

     hello('Bob')
     hello('Fred', do_something=True, loud=False)
     hello('Fred', True, False)   # Error
```

```
Hello, Bob!
Hello, Fred!
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-122-9f90866fb3e3> in <module>
      7 hello('Bob')
      8 hello('Fred', do_something=True, loud=False)
----> 9 hello('Fred', True, False)   # Error

TypeError: hello() takes 1 positional argument but 3 were given
```

# Syntax overview – classes

```
[ ] class Greeter:

        # Constructor
        def __init__(self, name: str):
            self._name = name    # Create an instance variable

        # Instance method
        def greet(self, loud=False):
            if loud:
                print('HELLO, %s!' % self._name.upper())
            else:
                print('Hello, %s' % self._name)

        # Property method
        @property
        def name(self) -> str:
            return self._name

        @property
        def initial(self) -> str:
            return "".join(v[0] for v in self.name.split())
```

```
[ ] g = Greeter('John Doe')  # Construct an instance of the Greeter class
    g.greet()             # Call an instance method; prints "Hello, John Doe"
    g.greet(loud=True)    # Call an instance method; prints "HELLO, JOHN DOE!"

    print(g.name)         # Call an instance property; prints "John Doe"
    print(g.initial)      # prints "JD"
    #g.name = 'Jane Doe'  # You cannot change the value of a property (unless you implement a 'setter')
```

```
Hello, John Doe
HELLO, JOHN DOE!
John Doe
JD
```

# How should you write Python code?

- OOP (object-oriented programming) is typical in ML
  - Models have all their functionality wrapped in a class
- Find the balance between having disorganized blobs of code versus wasting time overengineering everything
  - May take some practice!
  - For homeworks, not a huge deal as long as we can understand it
  - But for projects, you will probably want to put a little more thought into it
- For more detailed suggestions, see:
  - The Hitchhiker's Guide to Python: https://docs.python-guide.org/intro/learning/
  - Style Guide for Python Code (PEP-8): https://www.python.org/dev/peps/pep-0008/
  - Google Python Style Guide: https://google.github.io/styleguide/pyguide.html

# Packages/modules are code that you can import into your program

- They can be code in your project in a different location, or they can be code you have to fetch online with pip / Conda
  - (or just import it directly if you have a Colab notebook set up)
- NumPy, MatPlotLib, SciKit-Learn, PyTorch / TensorFlow are all packages

```
import torch
import numpy as np # aliasing
import tensorflow as tf # aliasing

torch.tensor([1,2])
np.array([0, 1])
mnist = tf.keras.datasets.mnist
```

# Outline

1. Introduction
2. Setup
3. Python
4. **NumPy**
5. MatPlotLib
6. Homework tips
7. Acknowledgements

# NumPy is the foundational scientific computing library for Python

- Under the hood, it's just a lot of matrix math coded in C++ to speed up computations
  - C++ runs faster than Python, so it makes sense to write a dedicated computing library in C++

# Main object: the NumPy array

```
[ ]  a = np.array([1, 2, 3])  # Create a rank 1 array
     print(type(a), a.shape, a[0], a[1], a[2])
     a[0] = 5                 # Change an element of the array
     print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

```
[ ]  b = np.array([[1, 2, 3], [4, 5, 6]])   # Create a rank 2 array
     b
     print(b.shape)
```

```
(2, 3)
```

```
[ ]  print(b[0, 0])   # preferred than b[0][0]
     print(b[0, 1])
```

```
1
2
```

Be aware of a **different type of slice indexing**:

```
[ ]  # Create the following rank 2 array with shape (3, 4)
     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
     a
```

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

```
[ ]  row_r1 = a[1, :]    # Rank 1 view of the second row of a
     row_r2 = a[1:4, :]  # Rank 2 view of the second row of a
     row_r3 = a[[1], :]  # Rank 2 view of the second row of a
     print(row_r1, row_r1.shape)
     print(row_r2, row_r2.shape)
     print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[ 5  6  7  8]
 [ 9 10 11 12]] (2, 4)
[[5 6 7 8]] (1, 4)
```

```
[ ]  # We can make the same distinction when accessing columns of an array:
     col_r1 = a[:, 1]
     col_r2 = a[:, 1:2]
     print(col_r1, col_r1.shape)
     print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Useful for getting rows / columns at a time instead of individual items

# Useful properties and functions

- NumPy array properties (call with array.<property_name>):
  - shape: return dimensions of array as a tuple
  - dtype: return datatype of array
- NumPy array functions (call with array.<function_name>() ):
  - reshape(<tuple of dimensions>): reshape array to specified dimensions
- NumPy Functions:
  - np.zeros(<tuple of dimensions>): create matrix of 0s with specified dimensions
  - np.random.random(<tuple of dimensions>): same as above, but fill with random vals in (0,1)
  - np.eye(<number n>): create (n,n) square identity matrix
- If you need some array operation, see if NumPy has it implemented already before writing your own impl

# NumPy supports matrix arithmetic as well

- For matching dimension matrices, perform element-wise operations
  - Except for matrix multiplication
  - Note matrix multiplication functions can serve as dot product too – this isn't universal

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])
```

```
# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
[ ] # Elementwise product; both produce the array
    print(x * y)
    print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
[ ] # Matrix / vector product; both produce the rank 1 array [29 67]
    print(x.dot(v))
    print(np.dot(x, v))
    print(x @ v)
```

```
[ ] # Matrix / matrix product; both produce the rank 2 array
    # [[19 22]
    #  [43 50]]
    print(x.dot(y))
    print(np.dot(x, y))
    print(x @ y)
```

- Mismatching dimensions will try to broadcast, then error out

# Other useful math functions:

- x.sum()
  - Specify an axis argument to choose which axis to sum along

```
[ ] x = np.array([[1,2,3], [4,5,6]])  # shape: (2, 3)

    print(np.sum(x))  # Compute sum of all elements; prints "21"
    print(np.sum(x, axis=0))  # Compute sum of each column; shape: (3), prints "[5 7 9]"
    print(np.sum(x, axis=1))  # Compute sum of each row;    shape: (2), prints "[6 15]"

    21
    [5 7 9]
    [ 6 15]
```

- x.T gives the transpose of x

```
[ ] # Transpose
    x.T

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

# NumPy broadcasting

- Essentially, implicit per-row / per-column operations

```
x =
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
v =
 [1 0 1]
```

```
[ ]  print(f"x.shape = {x.shape}, v.shape={v.shape}")
     y = x + v  # Add v to each row of x using broadcasting
     y

     x.shape = (4, 3), v.shape=(3,)
     array([[ 2,  2,  4],
            [ 5,  5,  7],
            [ 8,  8, 10],
            [11, 11, 13]])
```

- Use broadcasting where you can, especially in simpler HW problems
  - But if your code works, it's not necessarily worth engineering it to use broadcasting

# Aside: operations to extract rows / cols will usually return something 1D

- In 1D, they are indistinguishable
- I prefer keeping everything 2D to avoid confusion
  - But this is effort and adds memory overhead

```
[21] t = np.random.random((3,4))

     # if t were declared 100 lines above, how would you know this was a row vector?
     row = t[0, :]
     # if t were declared 100 lines above, how would you know this was a col vector?
     col = t[:, 0]
     print(row.shape, row)
     print(col.shape, col)

     # Suggested solution: write both in 2D
     row2 = t[0, :].reshape(1, 4)
     col2 = t[:, 0].reshape(3, 1)
     print(row2.shape, row2)
     print(col2.shape, col2)
```

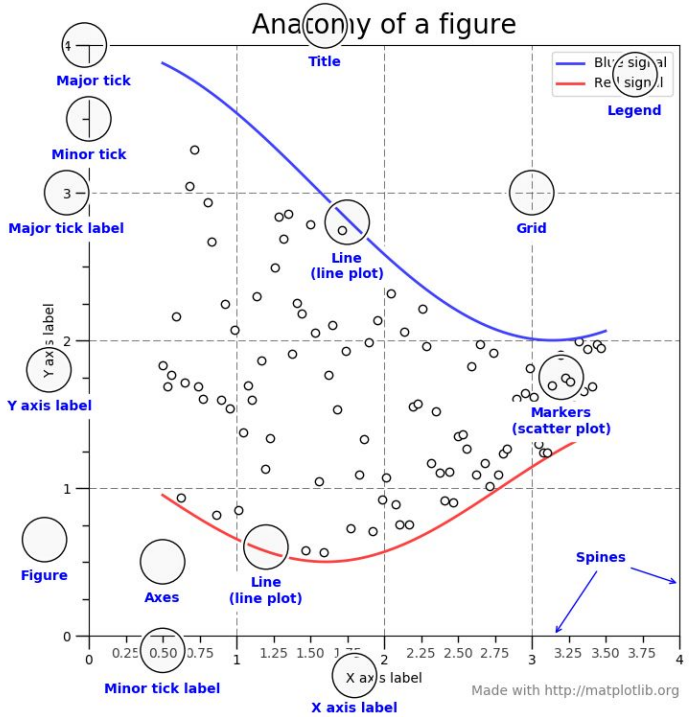
```
(4,) [0.09266646 0.89037841 0.88500717 0.99570662]
(3,) [0.09266646 0.01887809 0.42130118]
(1, 4) [[0.09266646 0.89037841 0.88500717 0.99570662]]
(3, 1) [[0.09266646]
 [0.01887809]
 [0.42130118]]
```

# Outline

1. Introduction
2. Setup
3. Python
4. NumPy
5. **MatPlotLib**
6. Homework tips
7. Acknowledgements

# MatPlotLib is a plotting tool

- Produces many of the nice charts you see in papers
- See right for a reference of what's what in a chart



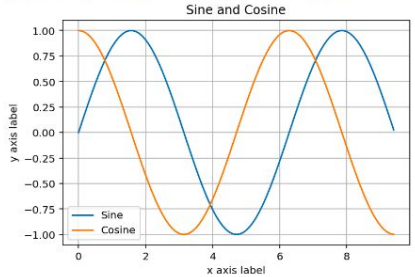
# MatPlotLib can be used globally or in an OOP way

- OOP is better stylistically, which matters more the larger your project is

```
[ ] 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.grid()
    plt.legend(['Sine', 'Cosine'])
```

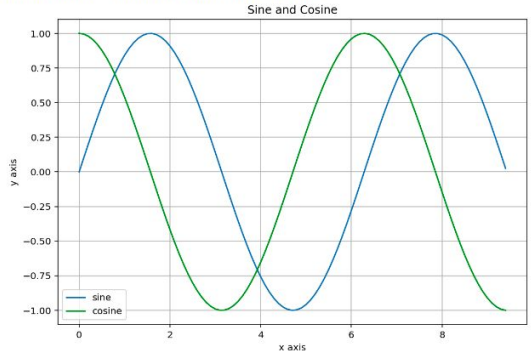
```
<matplotlib.legend.Legend at 0x13990b340>
```




Global versus OOP

```
[ ] fig, ax = plt.subplots(figsize=(9, 6))
    ax.plot(x, y_sin)
    ax.plot(x, y_cos, color='green')
    ax.set_xlabel('x axis')
    ax.set_ylabel('y axis')
    ax.set_title('Sine and Cosine')
    ax.grid()
    ax.legend(['sine', 'cosine'])
```

```
<matplotlib.legend.Legend at 0x139965f70>
```



# More elaborate walkthrough is in the notebook and online doc pages

- Getting the results is more important than making them artistic, so I'm skimming the details here

# Outline

1. Introduction
2. Setup
3. Python
4. NumPy
5. MatPlotLib
6. **Homework tips**
7. Acknowledgements

# We run your homework code as Python scripts, so submit files

- But feel free to work in a notebook
  - If you do, note the following lets you execute code in a notebook from a .py module:

```
[ ]  %run q1.py
```

```
[ ]  %load_ext autoreload
     %autoreload 2

     import q1
     q1.stochastic_gradient_descent(X_train=[], y_train=[])
```

# Example HW structure

```
[ ]  # Example of q1.py
     # Run the program: $ python3 q1.py

     import os
     import numpy as np

     def batch_gradient_descent(X_train, y_train, learning_rate=1e-3):
         # ... do some computation ...
         return w, some_additional_data


     def stochastic_gradient_descent(X_train, y_train):
         # ...
         raise NotImplementedError()


     def main():
         X_train = np.load('q1xTrain.npy')
         y_train = np.load('q1yTrain.npy')

         w, info = batch_gradient_descent(X_train, y_train, ...)
         plot_result(w, info)


     if __name__ == '__main__':
         main()
```

# Outline

1. Introduction
2. Setup
3. Python
4. NumPy
5. MatPlotLib
6. Homework tips
7. **Acknowledgements**

# Shoutouts to these folks!

- Jongwook Choi, EECS 545 W20 Python Tutorial
- Justin Johnson, CS231n Python/NumPy Tutorial
- Brandon Nyugen, EECS 201 Python lectures
- Stack Overflow, W3Schools, Real Python, and other great websites for when you forget how something basic works
- Too many programming blogs to name