Software Engineering Laboratory CS29006

Abir Das

Computer Science and Engineering Department
Indian Institute of Technology Kharagpur

Agenda

- Getting familiar with some efficient tools in Python
- Getting familiar with OOP concepts in Python
- Getting familiar with numpy and matplotlib
- Getting to know what is segmenting objects in images
- Assumption: You are already familiar with basics of Python e.g., conditions, loops, functions, different containers.

- All codes used in this slide as well as codes to get you started on the assignments are in the public github repo: https://github.com/dasabir/CS29006_SW_Lab_Spr2022

Getting Started

- **Running Python:**
  - There are many ways to install Python on your laptop/PC/server etc.
    - https://www.python.org/downloads/
    - https://www.anaconda.com/download/
  - There are many editors as well
    - Eclipse
    - Jupyter notebook/lab
    - Spyder
    - PyCharm
    - VSCode
    - Text editors like Sublime Text

Our Choice is Popular

- **Anaconda:**
  - Anaconda is a distribution of programs in Python (and R) language and includes a huge number of libraries and several tools.
  - These include the Spyder development environment and Jupyter notebooks.
  - You can create your own customized environment which is independent of what you have in your PC/Laptop/Server already.

Installing Anaconda

- Download and install Anaconda for your OS -- https://www.anaconda.com/products/individual#Downloads -- Note that the most recent python version is 3.9 here. There are two versions of the installer – **Graphical** and **Commandline**. Graphical works on windows/mac while Commandline works on mac/Linux [In the above link the linux specific installer is called just 'installer' [i.e., without the word 'commandline' in it]]. If you have the option, use **Commandline** installation [that will be my choice].
- Depending on your OS please follow the steps as listed in the following links.
  - Installing on Windows
  - Installing on macOS
  - Installing on Linux

Installing Anaconda

- Straightaway after installing, you can use spyder and jupyter notebook ides. Get started from -- https://docs.anaconda.com/anaconda/user-guide/getting-started/

- If you can run till this, you are ready for the lab! However I shall continue to use eclipse editor [Sorry – comfort zone]. In eclipse you need to add PyDev plugin -- https://docs.anaconda.com/anaconda/user-guide/tasks/integration/eclipse-pydev/

Sources

- Materials for these slides are taken majorly from the following websites.

  - https://www.thedigitalcatonline.com/blog/2014/08/20/python-3-oop-part-1-objects-and-types/
  - https://www.pythonlikeyoumeanit.com/intro.html
  - https://cs231n.github.io/python-numpy-tutorial/

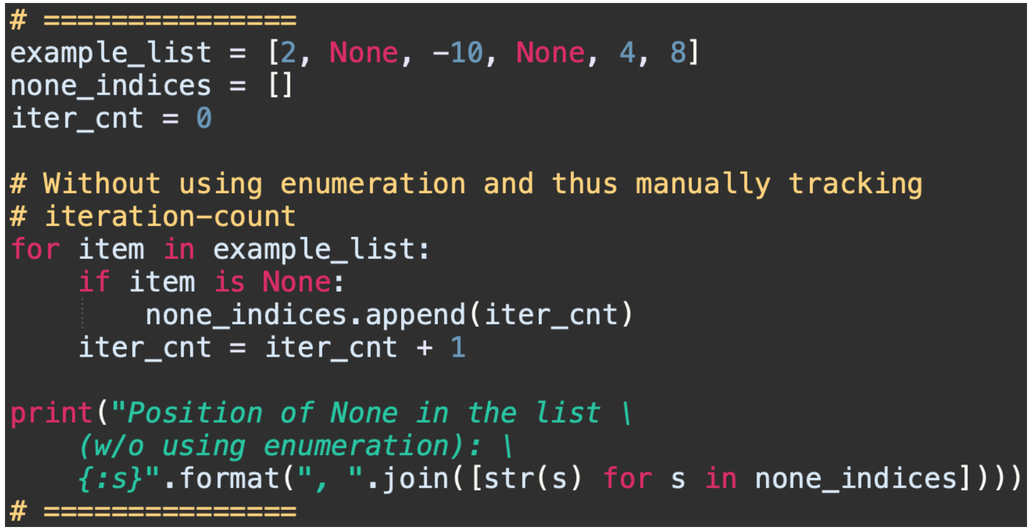
Iterables

- An **iterable** is any Python object capable of returning its members one at a time, permitting it to be iterated over a loop.
  - Examples:- lists, tuples, and strings etc.
  - Iterables help to write efficient codes using the concept of 'generators' – which we will come at a later slide.

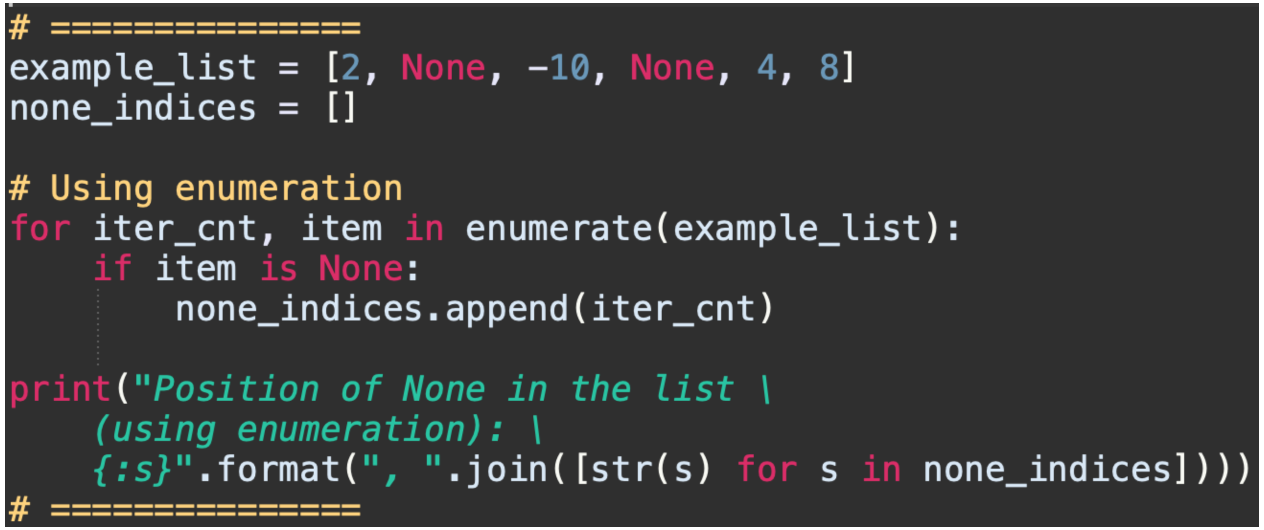
- Some useful built-in functions that accept iterables as arguments:
  - list, tuple, sum, sorted etc.
  - Demo time

Enumerating Iterables

- The built-in enumerate function allows to iterate over an iterable, while keeping track of the iteration count.
- The enumerate function accepts an iterable as an input and the items in the iterable are enumerated.
- Example code illustrating simplification of code. Problem statement is – to record all the positions in a list where the value None is stored.



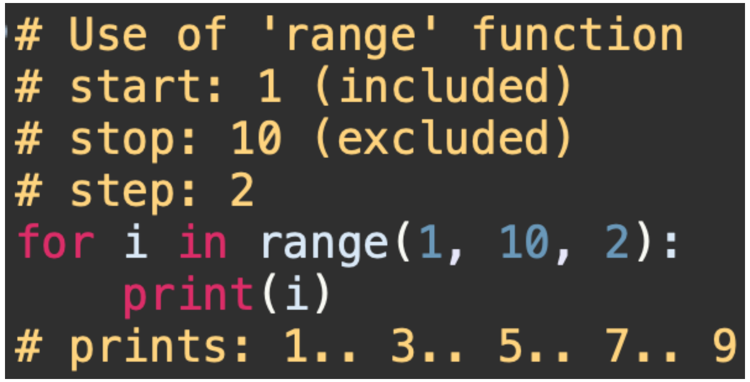
```
# ===============
example_list = [2, None, -10, None, 4, 8]
none_indices = []
iter_cnt = 0

# Without using enumeration and thus manually tracking
# iteration-count
for item in example_list:
    if item is None:
        none_indices.append(iter_cnt)
    iter_cnt = iter_cnt + 1

print("Position of None in the list \
    (w/o using enumeration): \
    {:s}".format(", ".join([str(s) for s in none_indices])))
# ===============
```



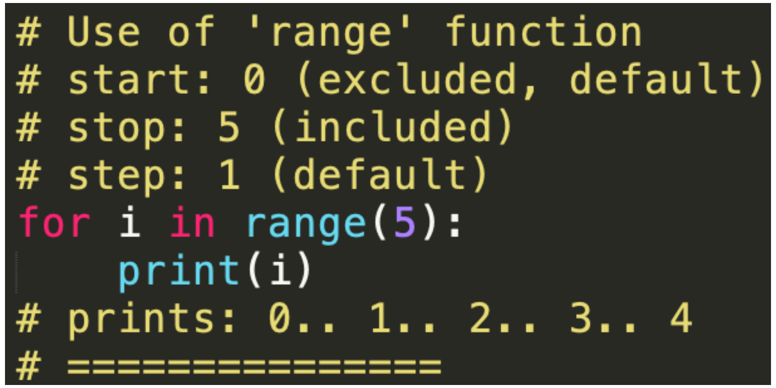
```
# ================
example_list = [2, None, -10, None, 4, 8]
none_indices = []

# Using enumeration
for iter_cnt, item in enumerate(example_list):
    if item is None:
        none_indices.append(iter_cnt)

print("Position of None in the list \
    (using enumeration): \
    {:s}".format(", ".join([str(s) for s in none_indices])))
# ================
```

Generators

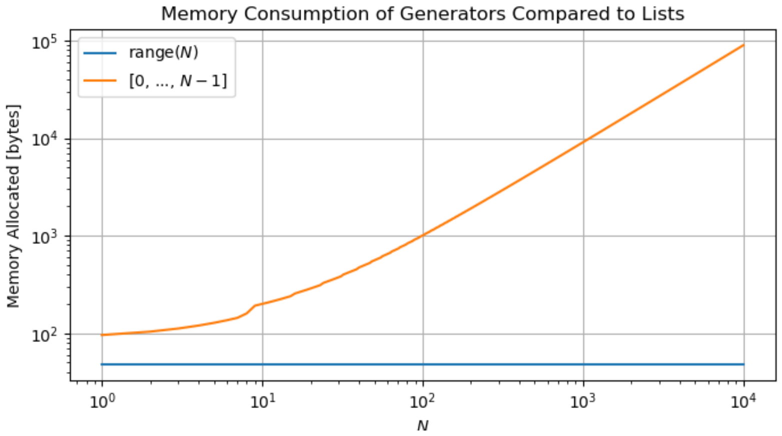
- Generators allow us to generate arbitrarily-many items in a series, without having to store them all in memory at once.
- Recall that a list readily stores all of its members. A generator, on the other hand, stores the instructions for generating each of its members, and stores its iteration state; this means that the generator will know if it has generated its second member, and will thus generate its third member the next time it is iterated on.



```
# Use of 'range' function
# start: 1 (included)
# stop: 10 (excluded)
# step: 2
for i in range(1, 10, 2):
    print(i)
# prints: 1.. 3.. 5.. 7.. 9
```

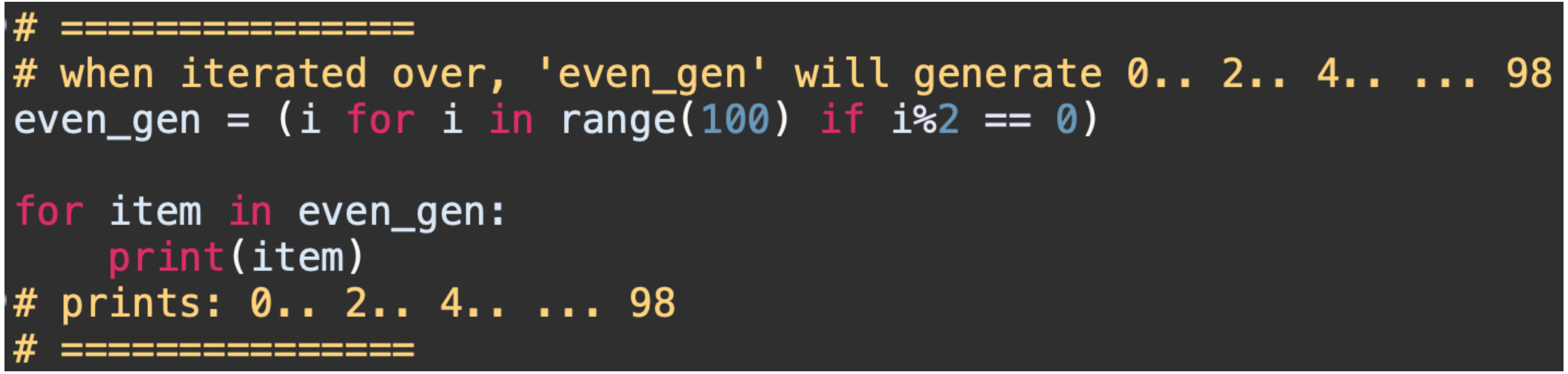


```
# Use of 'range' function
# start: 0 (excluded, default)
# stop: 5 (included)
# step: 1 (default)
for i in range(5):
    print(i)
# prints: 0.. 1.. 2.. 3.. 4
# ===============
```



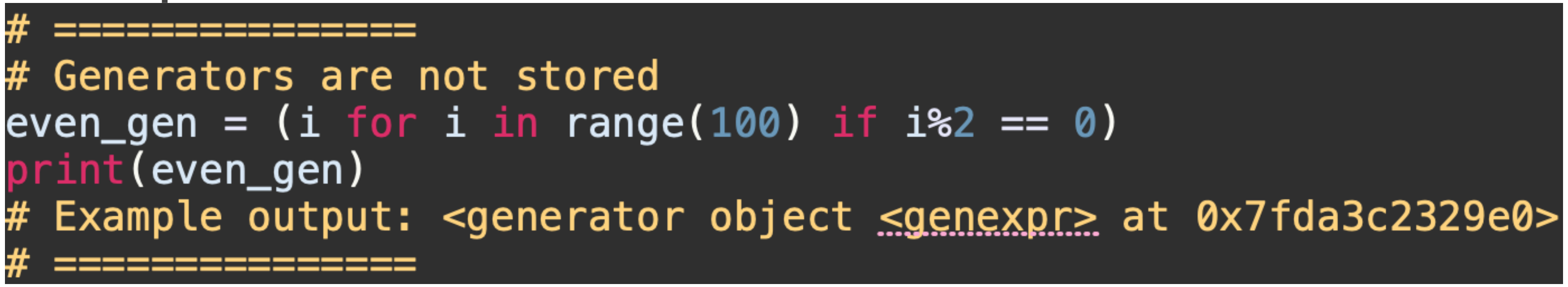
Generator Comprehensions

- Python provides a sleek syntax for defining a simple generator in a single line of code – known as Generator Comprehension
- The syntax is - (<expression> for <var> in <iterable> [if <condition>])
- (<expression> can be any valid single-line of Python code that returns an object:



```
# ===============
# when iterated over, 'even_gen' will generate 0.. 2.. 4.. ... 98
even_gen = (i for i in range(100) if i%2 == 0)

for item in even_gen:
    print(item)
# prints: 0.. 2.. 4.. ... 98
# ===============
```

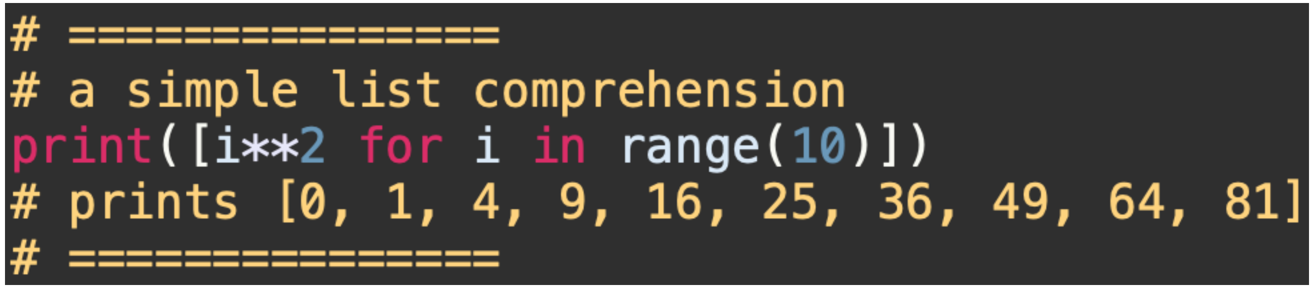
- Generator comprehensions do not store values.



```
# ===============
# Generators are not stored
even_gen = (i for i in range(100) if i%2 == 0)
print(even_gen)
# Example output: <generator object <genexpr> at 0x7fda3c2329e0>
# ===============
```

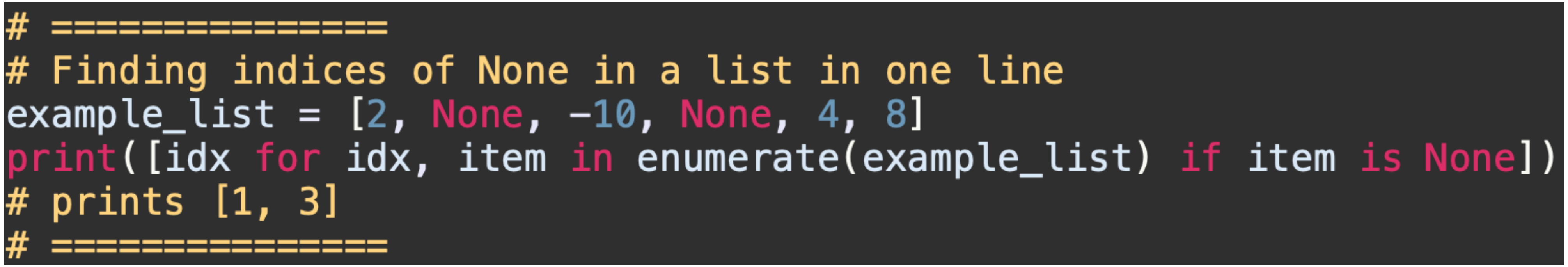
List (and Tuple) Comprehensions

- Using generator comprehensions to initialize lists is so useful that Python actually reserves a specialized syntax for it, known as the list comprehension
- The syntax is: [<expression> **for** <var> **in** <iterable> {**if** <condition}]



```
# ===============
# a simple list comprehension
print([i**2 for i in range(10)])
# prints [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# ===============
```

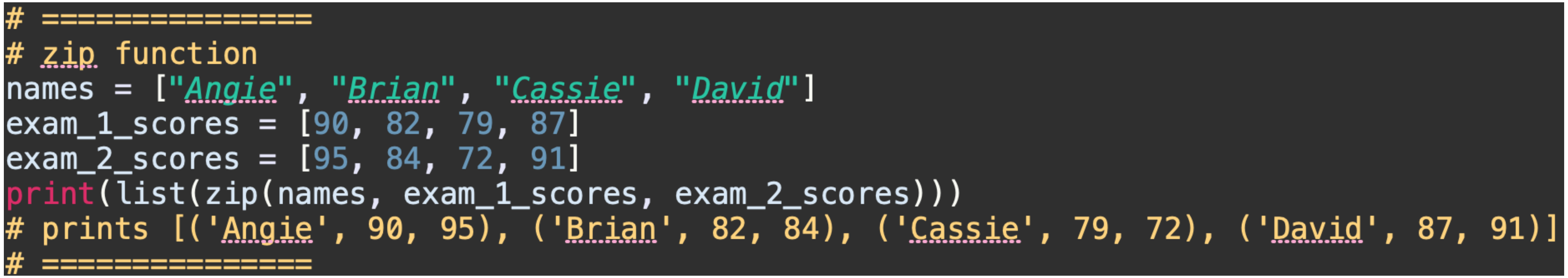
- Revisiting the example of finding index of ‘None’ in a list



```
# ===============
# Finding indices of None in a list in one line
example_list = [2, None, -10, None, 4, 8]
print([idx for idx, item in enumerate(example_list) if item is None])
# prints [1, 3]
# ===============
```

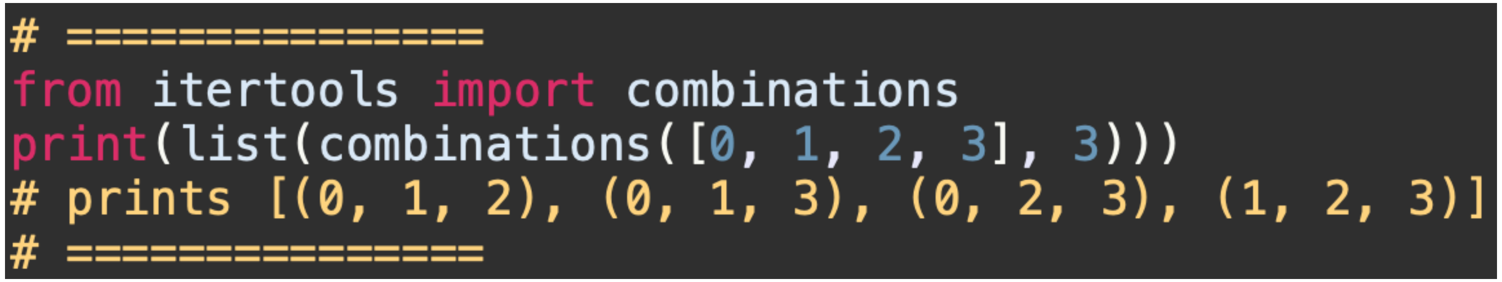
Itertools

- Python has an **itertools** module, which provides a core set of fast, memory-efficient tools for creating iterators. The majority of these functions create generators, thus we will have to iterate over them in order to show the use of them.
- zip: Zips together the corresponding elements of several iterables into tuples.



```
# ===============
# zip function
names = ["Angie", "Brian", "Cassie", "David"]
exam_1_scores = [90, 82, 79, 87]
exam_2_scores = [95, 84, 72, 91]
print(list(zip(names, exam_1_scores, exam_2_scores)))
# prints [('Angie', 90, 95), ('Brian', 82, 84), ('Cassie', 79, 72), ('David', 87, 91)]
# ===============
```

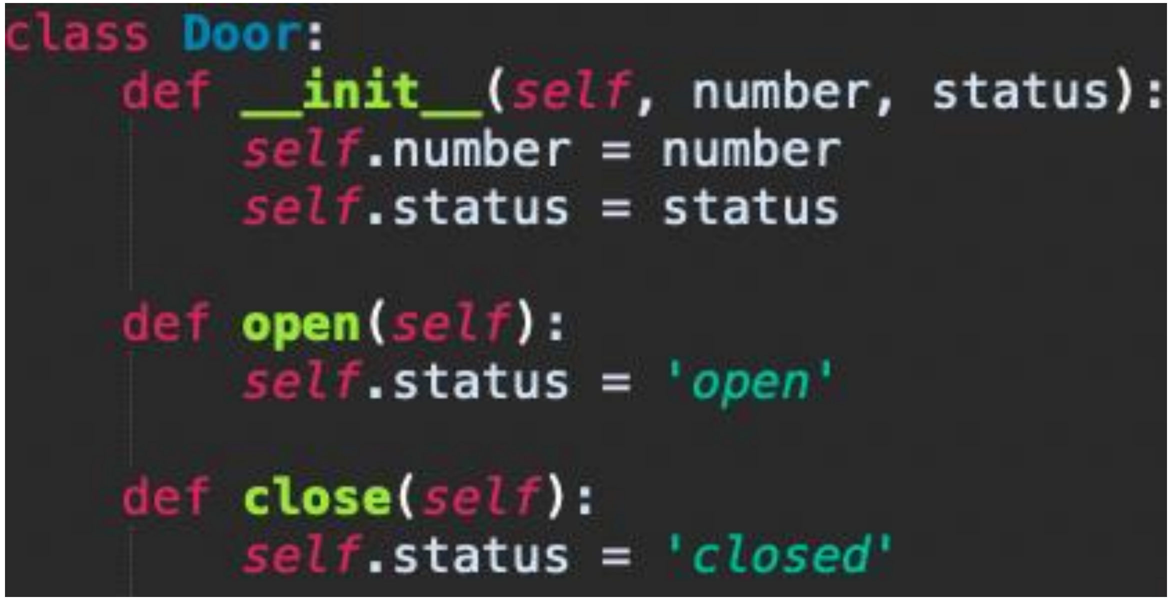
- itertools.combinations: Generate all length-n tuples storing “combinations” of items from an iterable:



```
# ===============
from itertools import combinations
print(list(combinations([0, 1, 2, 3], 3)))
# prints [(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
# ===============
```

Object Oriented Programing in Python

- We will discuss some key terminology for object-oriented programming in python. Most references will be made along the topics getting covered in the theory class.
- The class keyword is reserved for defining a class.
- The following defines a new class of object, named Door, specifying two attributes number and status, and two member functions open and close.



```python
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

Creating Objects

- Methods of a class must accept as first argument a special value called self (the name is a convention but please never break it).
- The special method __init__() works as the constructor.
- door1 = Door(1, 'closed’)
- print(door1.number) # gives 1
- print(door1.status) # gives closed
- door1.open()
    - No arguments have been passed. But, it was declared to accept an argument (self). When you call a method of an instance, the instance is passed to the method as first argument automatically.
- print(door1.status) # gives open
- print(type(door1)) # gives <class '__main__.Door’>
    - type() returns the class as __main__.Door since the class was defined directly in the interactive shell, that is in the current main module.

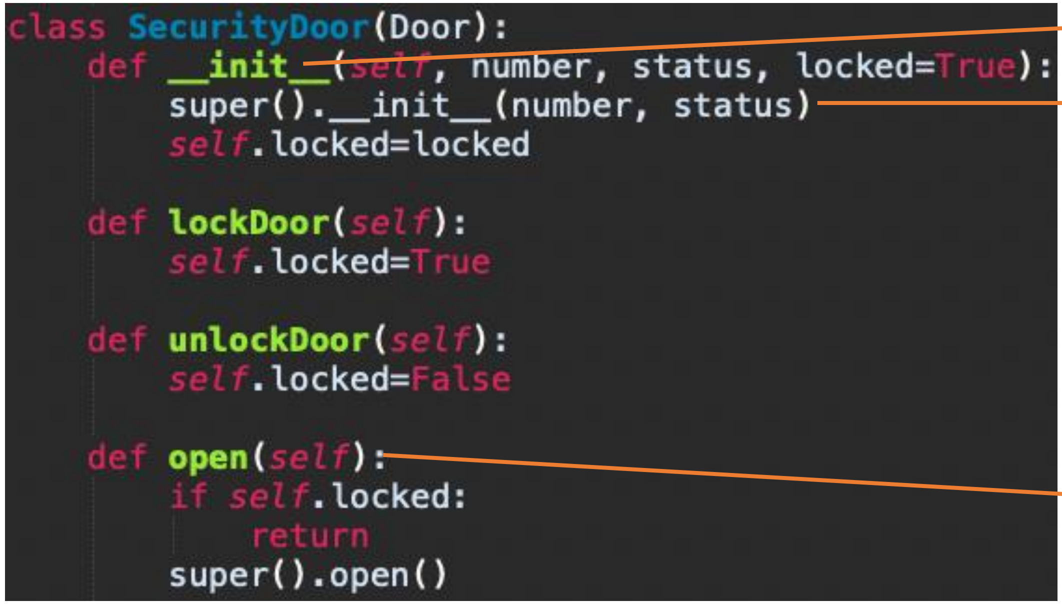
Playing with Addresses

- Create one more Door object with same attributes
- `door2 = Door(1, 'closed')`
- `print(hex(id(door1)))` # gives 0x7faebb2c1310
- `print(hex(id(door2)))` # gives 0x7faebb29e1c0
- `print(hex(id(door1.__class__)))` # gives 0x7faeb9d1de40
- `print(hex(id(door2.__class__)))` # gives 0x7faeb9d1de40

- Any Python object is automatically given a __dict__ attribute, which contains its list of attributes. Try both – "Door.__dict__" and "door1.__dict__"
- You can also get the attribute value by "door1.__dict__['status']" [try it]

Inheritance and Overriding

- Let us try to create a subclass of Door called SecurityDoor which has an additional attribute that provides the information whether the door is locked.



```
class SecurityDoor(Door):
    def __init__(self, number, status, locked=True):
        super().__init__(number, status)
        self.locked=locked

    def lockDoor(self):
        self.locked=True

    def unlockDoor(self):
        self.locked=False

    def open(self):
        if self.locked:
            return
        super().open()
```

Subclass constructor

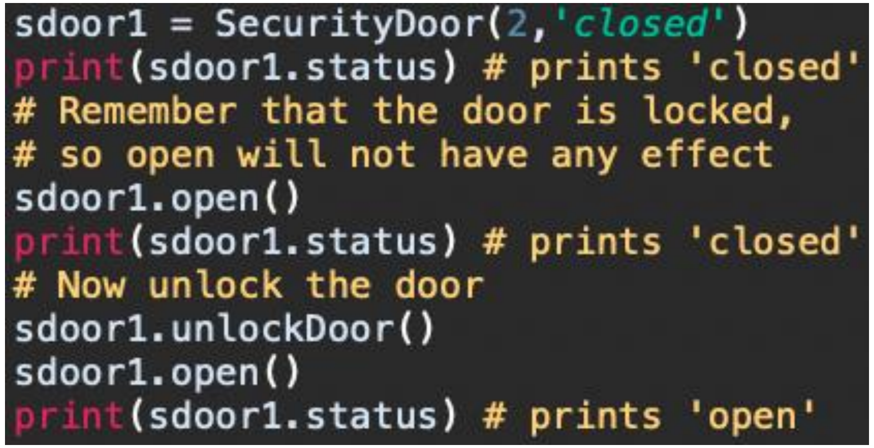
Calling superclass constructor

Override: In Python you can override a parent class member simply by redefining it in the child class.

- Instead of 'super' you could have used 'Door' [e.g., Door.__init__() or Door.open() etc]. However, using 'super' is encouraged as this lets python do the hierarchy resolution for multiple inheritance.

Inheritance and Overriding

- Behavior of an object of type SecurityDoor.

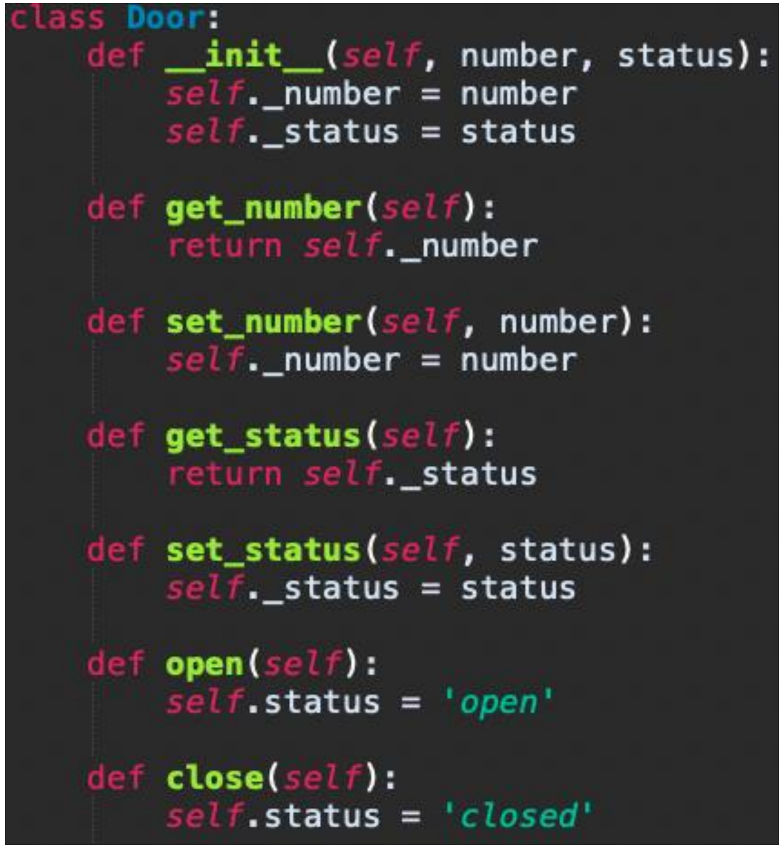


```
sdoor1 = SecurityDoor(2,'closed')
print(sdoor1.status) # prints 'closed'
# Remember that the door is locked,
# so open will not have any effect
sdoor1.open()
print(sdoor1.status) # prints 'closed'
# Now unlock the door
sdoor1.unlockDoor()
sdoor1.open()
print(sdoor1.status) # prints 'open'
```

- Try the following methods.
    - SecurityDoor.__bases__
    - Print(help(SecurityDoor))

Encapsulation in Python

- Python inherently does not force encapsulation. However, there are ‘pythonic’ conventions and ways to do it.



```python
class Door:
    def __init__(self, number, status):
        self._number = number
        self._status = status

    def get_number(self):
        return self._number

    def set_number(self, number):
        self._number = number

    def get_status(self):
        return self._status

    def set_status(self, status):
        self._status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

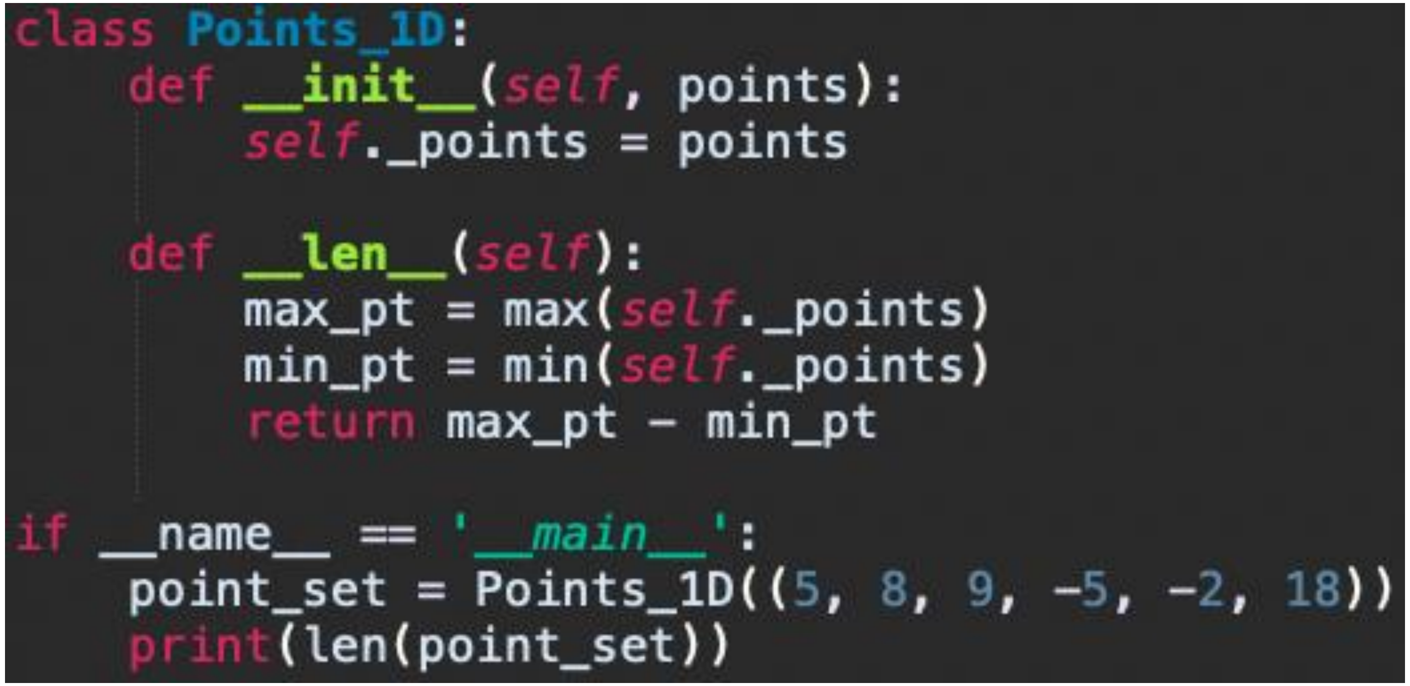
- Pythonic way – Use of property decorators – getters and setters [Good resource: Link]

Python Magic Methods

- Nothing magical about it. They are special methods with fixed names. These are a set of predefined methods you can use to enrich your classes.
- They are easy to recognize because of the double underscores at the beginning and the end.
- We have already encountered '__init__' method.
- "Underscore underscore init underscore underscore" is not easy going to pronounce. So, people say - "dunder init dunder" . 'dunder' is short form of 'double underscore'

- So, what's 'magic' about the magic or dunder methods? - The answer is, you don't have to invoke it directly. The invocation is realized behind the scenes. When you create an instance of a class, python makes the necessary call to the '__init__()' method.
- To get the length of a string/tuple/list etc. you can call len(). However, for an user defined class len may not work. You need to add a __len__() dunder method to fix this.

Python Magic Methods

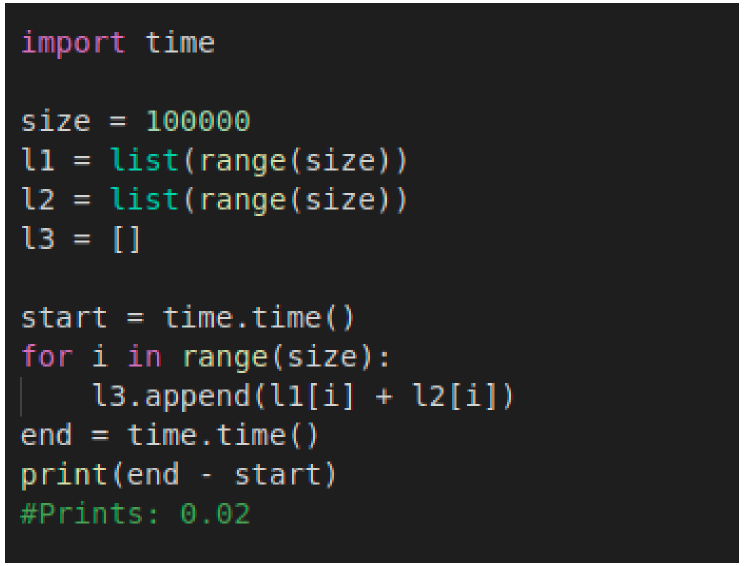


```python
class Points_1D:
    def __init__(self, points):
        self._points = points

    def __len__(self):
        max_pt = max(self._points)
        min_pt = min(self._points)
        return max_pt - min_pt

if __name__ == '__main__':
    point_set = Points_1D((5, 8, 9, -5, -2, 18))
    print(len(point_set))
```

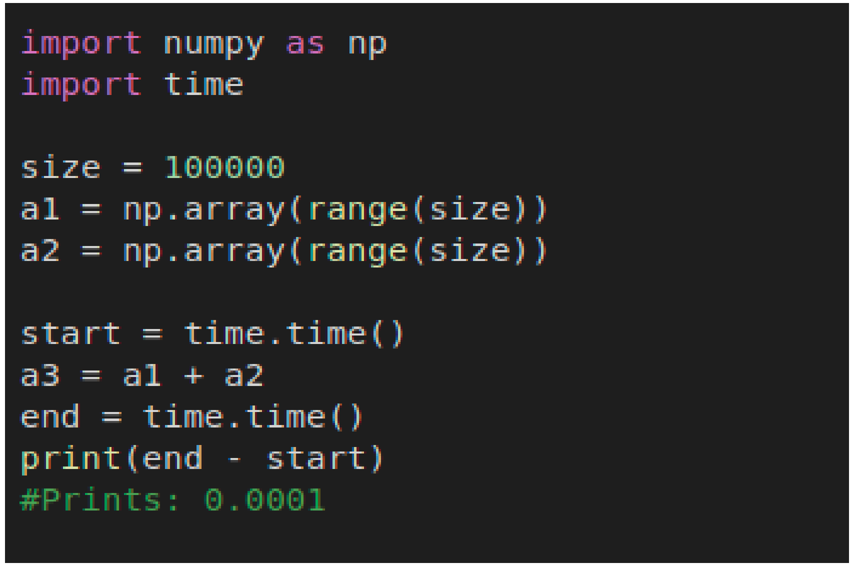
- The above prints the distance between the max and min values of the set of 1-D points.
- There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the __add__ method. For "-" it is __sub__ and so on.
- List of important magic methods - Link

Numpy

- Numpy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.



```python
import time

size = 100000
l1 = list(range(size))
l2 = list(range(size))
l3 = []

start = time.time()
for i in range(size):
    l3.append(l1[i] + l2[i])
end = time.time()
print(end - start)
#Prints: 0.02
```



```python
import numpy as np
import time

size = 100000
a1 = np.array(range(size))
a2 = np.array(range(size))

start = time.time()
a3 = a1 + a2
end = time.time()
print(end - start)
#Prints: 0.0001
```

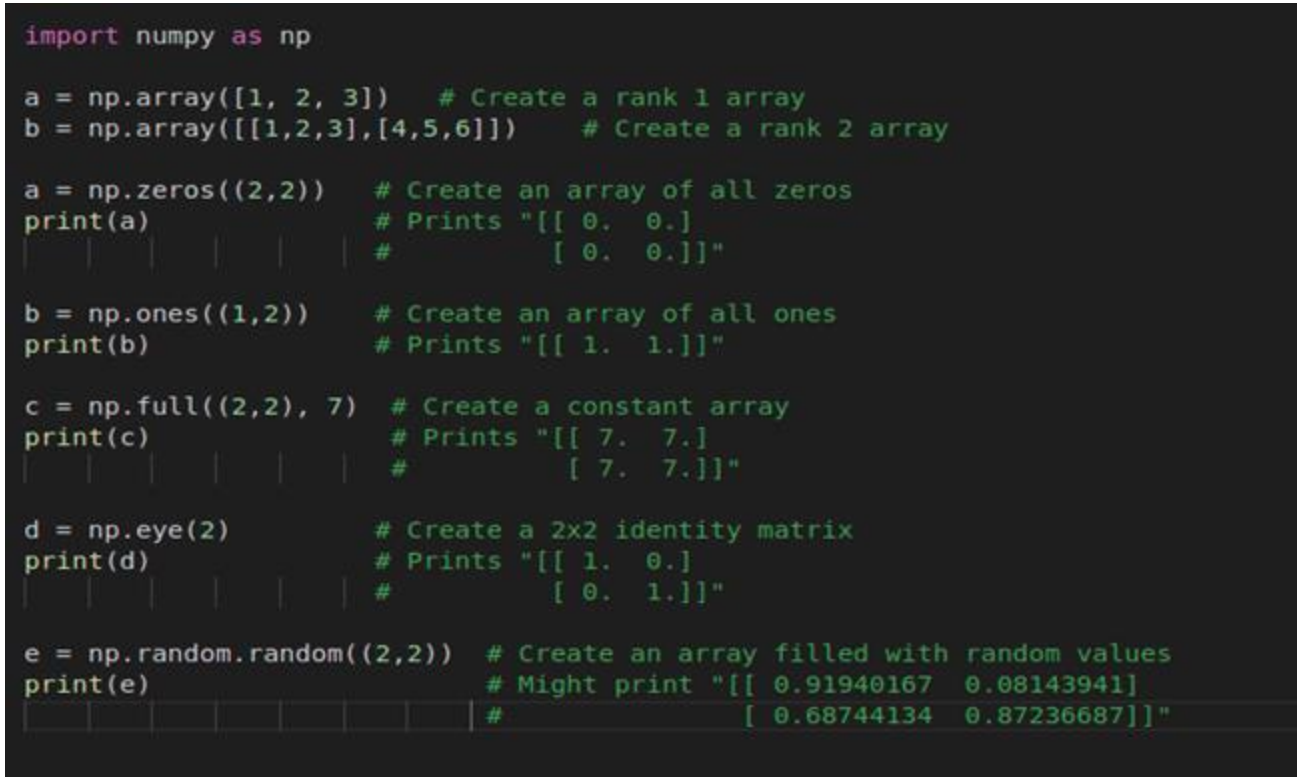
- Operating on numpy arrays are way faster than looping on lists. The trick is to replace all the loops by the vectorized operations allowed on numpy arrays.

ref: https://cs231n.github.io/python-numpy-tutorial/

Numpy Arrays

- A numpy array is a grid of values, **all of the same type**, and is indexed by a tuple of nonnegative integers. The number of dimensions is the **rank** of the array; the **shape** of an array is a tuple of integers giving the size of the array along each dimension.

- Use np.array() to create numpy arrays from lists.
- Use np.zeros(), np.ones(), np.full() etc to create numpy arrays with specific values.
- Use np.random.random() to create arrays with random numbers.



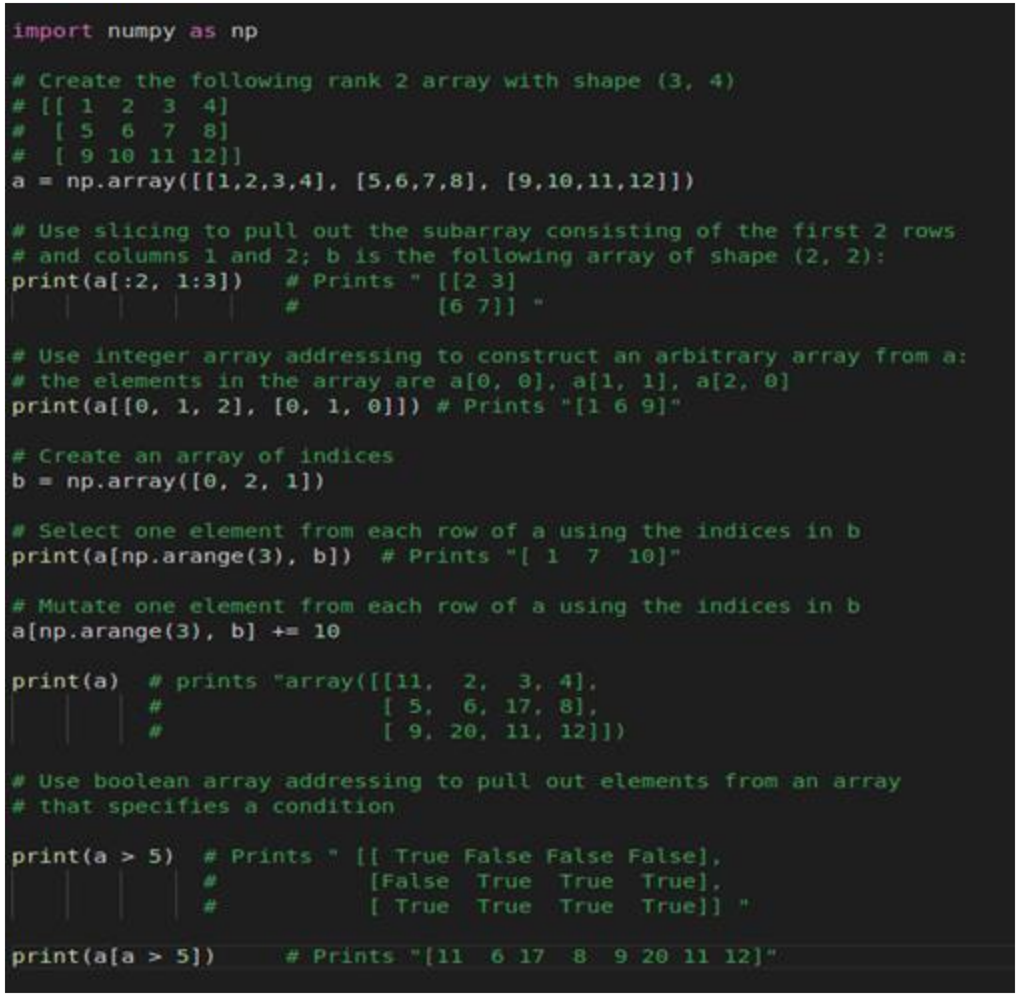
```
import numpy as np

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

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

ref: https://cs231n.github.io/python-numpy-tutorial/

Array Indexing

Numpy offers several ways to index into arrays,

- **Slicing**: Similar to Python lists, but you must specify a slice for each dimension of the array.
- **Integer Array Indexing**: This allows you to construct arbitrary arrays using the data from another array.
- **Boolean Array Indexing**: This type of indexing is used to select the elements of an array that satisfy some condition.

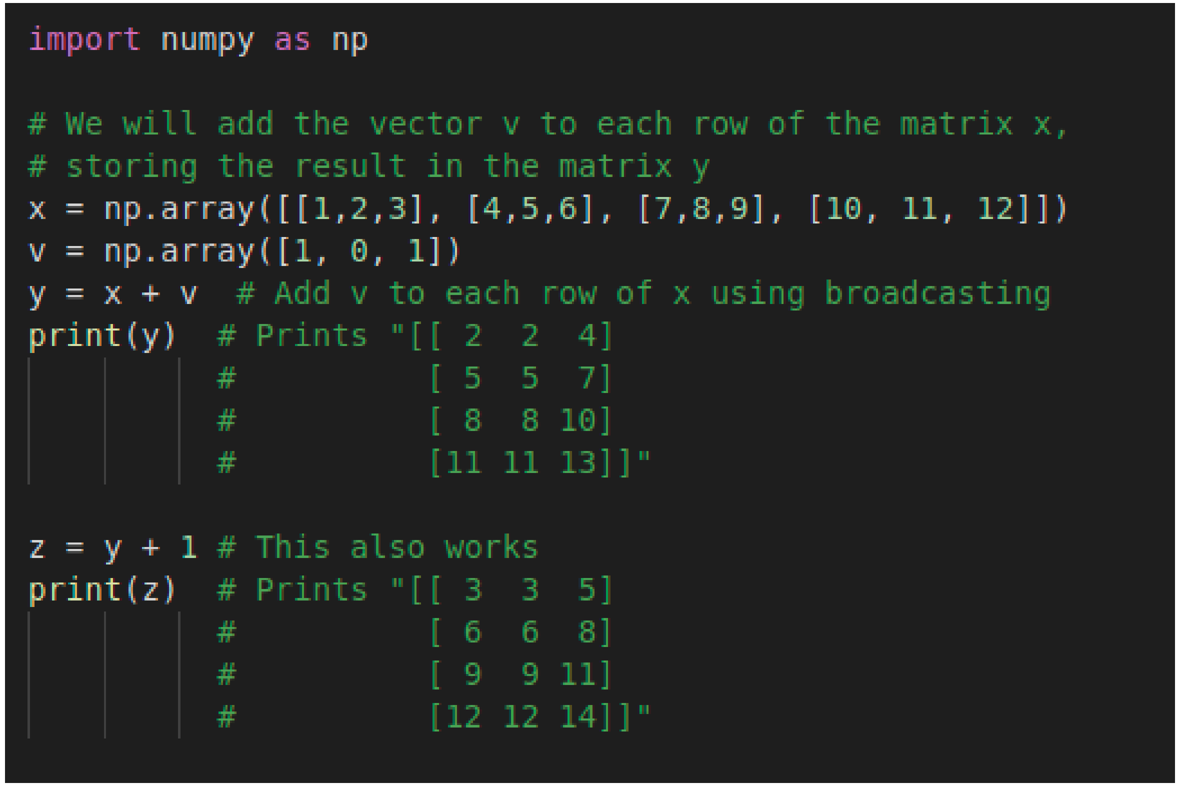


```
import numpy as np

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

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
print(a[:2, 1:3])   # Prints " [[2 3]
                    #           [6 7]] "

# Use integer array addressing to construct an arbitrary array from a:
# the elements in the array are a[0, 0], a[1, 1], a[2, 0]
print(a[[0, 1, 2], [0, 1, 0]]) # Prints "[1 6 9]"

# Create an array of indices
b = np.array([0, 2, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(3), b])  # Prints "[ 1  7  10]"

# Mutate one element from each row of a using the indices in b
a[np.arange(3), b] += 10

print(a)  # prints "array([[11,  2,  3, 4],
          #                [ 5,  6, 17, 8],
          #                [ 9, 20, 11, 12]])

# Use boolean array addressing to pull out elements from an array
# that specifies a condition

print(a > 5)  # Prints " [[ True False False False],
              #           [False  True  True  True],
              #           [ True  True  True  True]] "

print(a[a > 5])     # Prints "[11  6 17  8  9 20 11 12]"
```

ref: https://cs231n.github.io/python-numpy-tutorial/

Broadcasting

- Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.
- The line y = x + v works even though x has shape (4, 3) and v has shape (3,) due to broadcasting; this line works as if v actually had shape (4, 3), where each row was a copy of v, and the sum was performed elementwise.
- Similarly, 1 is added as it has been copied to all elements of an array of shape (4, 3).
- Broadcasting does not work on any arbitrary array shapes. It follows certain rules.
- A good article - [Link]

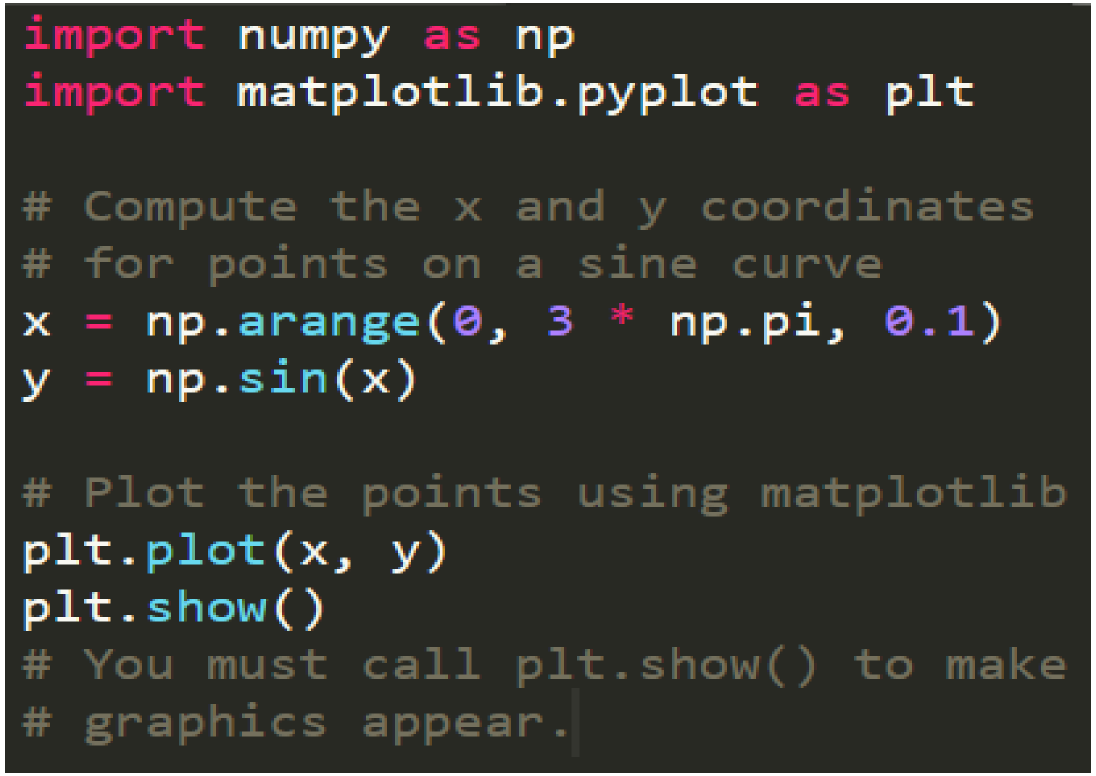


```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"

z = y + 1 # This also works
print(z)  # Prints "[[ 3  3  5]
          #          [ 6  6  8]
          #          [ 9  9 11]
          #          [12 12 14]]"
```

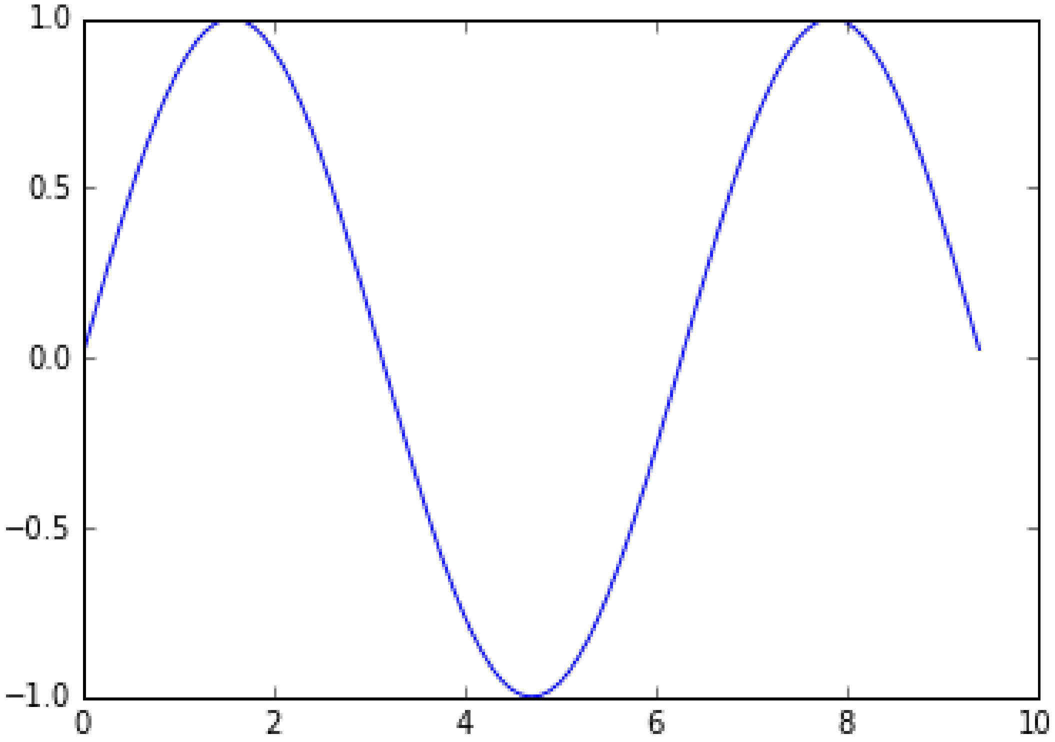
ref: https://cs231n.github.io/python-numpy-tutorial/

Matplotlib (Pyplot)

- Matplotlib is one of the most popular Python packages used for data visualization.
- The most important function is plot, which allows you to plot 2D data. Simple example:

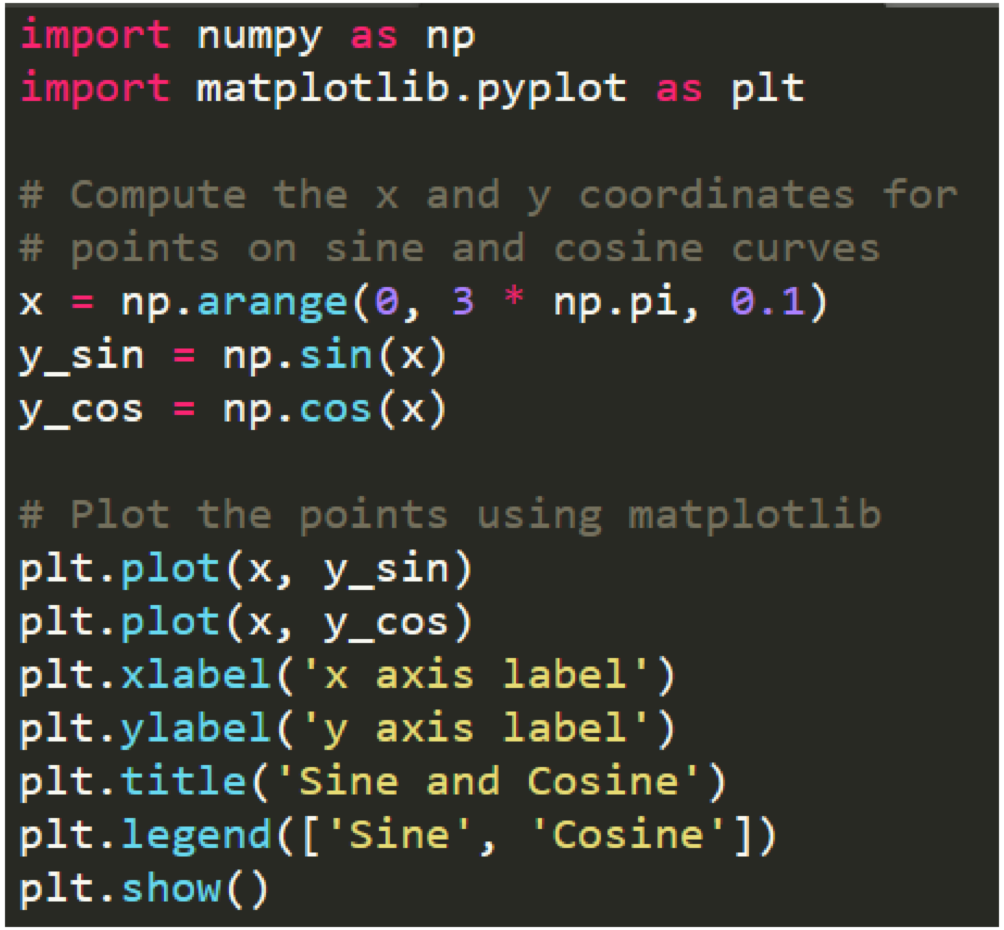


```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates
# for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
# You must call plt.show() to make
# graphics appear.
```

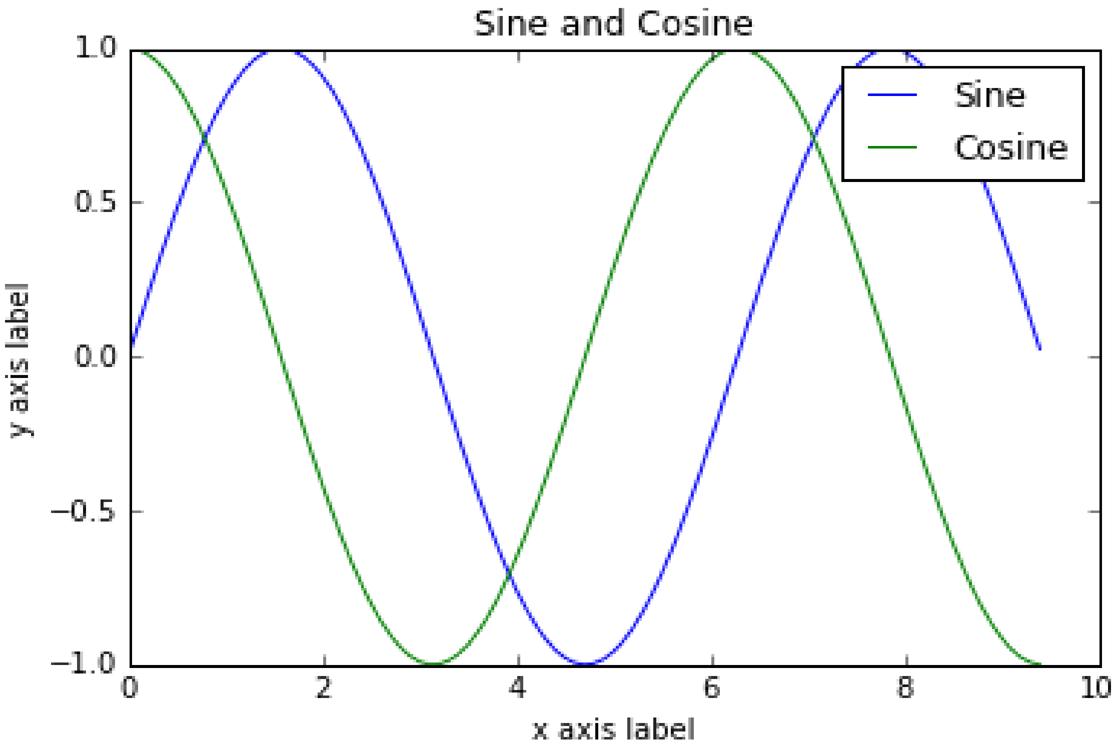


ref: https://cs231n.github.io/python-numpy-tutorial/

Multiple plots, legends, and axis labels

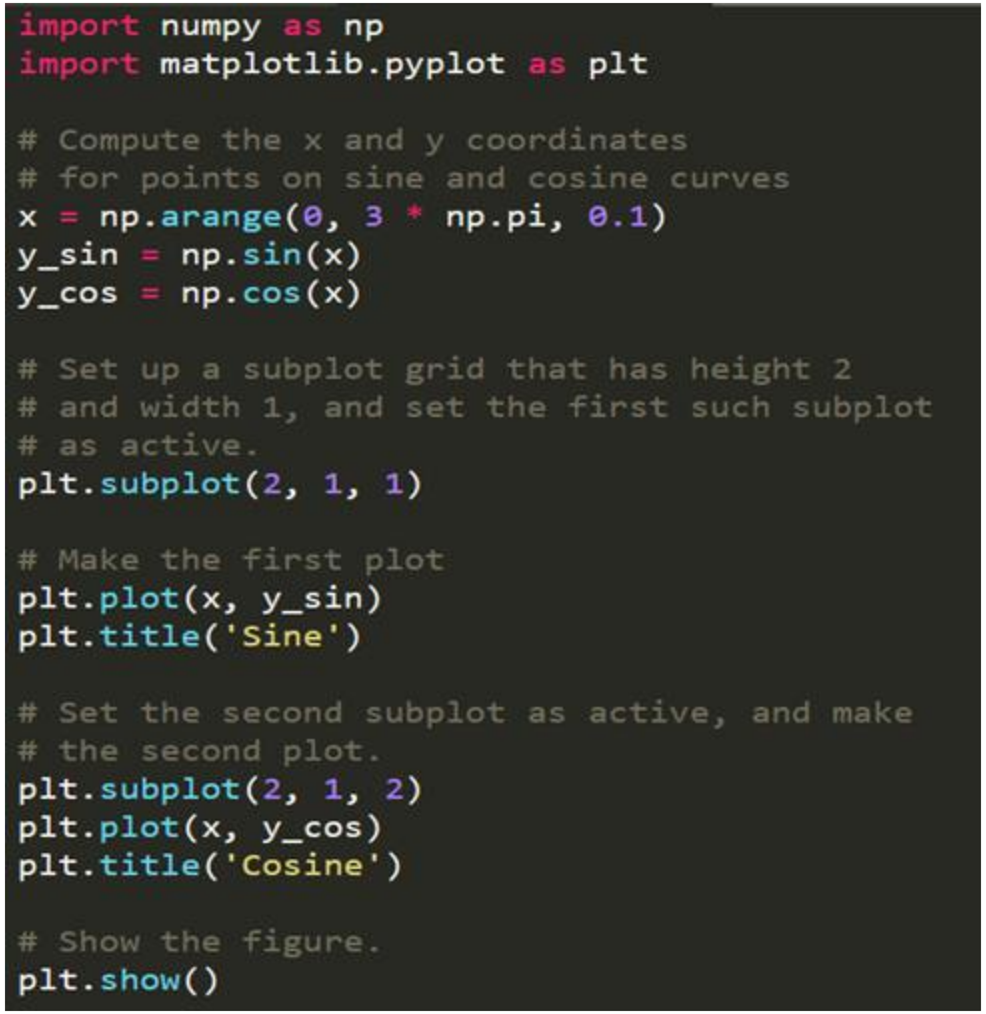


```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for
# points on sine and cosine curves
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()
```

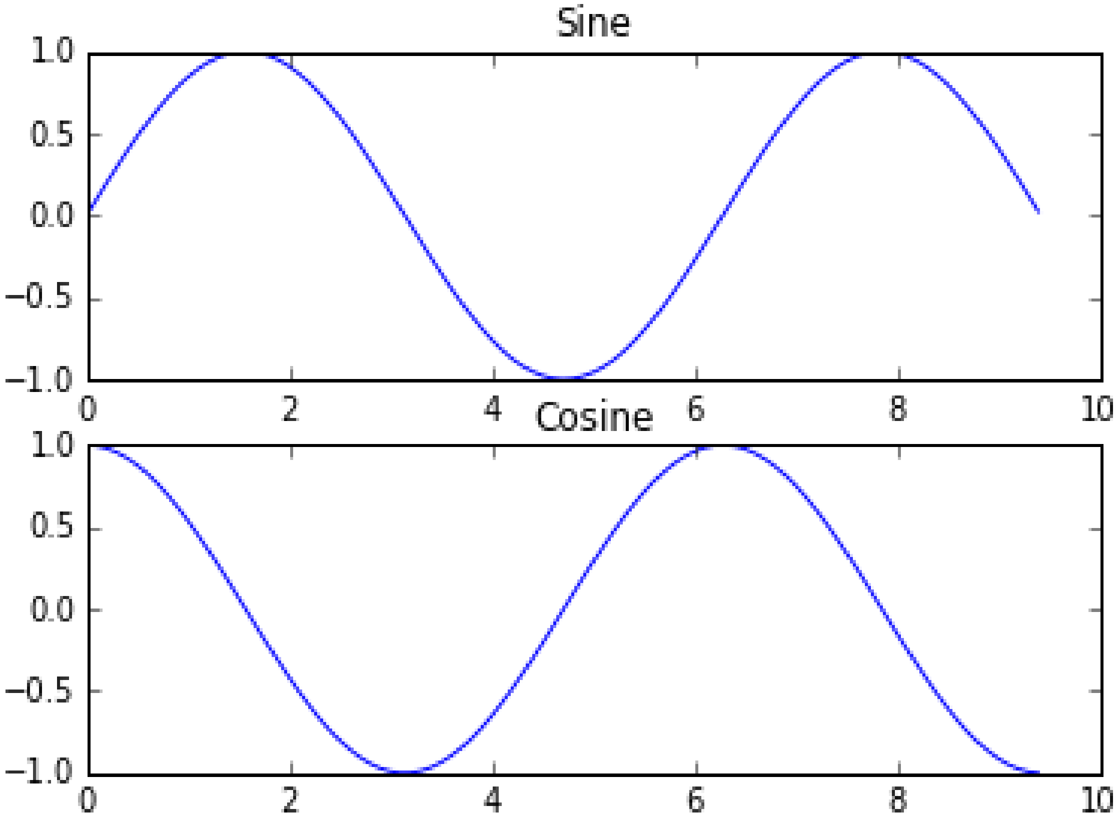


ref: https://cs231n.github.io/python-numpy-tutorial/

Subplots (different things in the same figure)



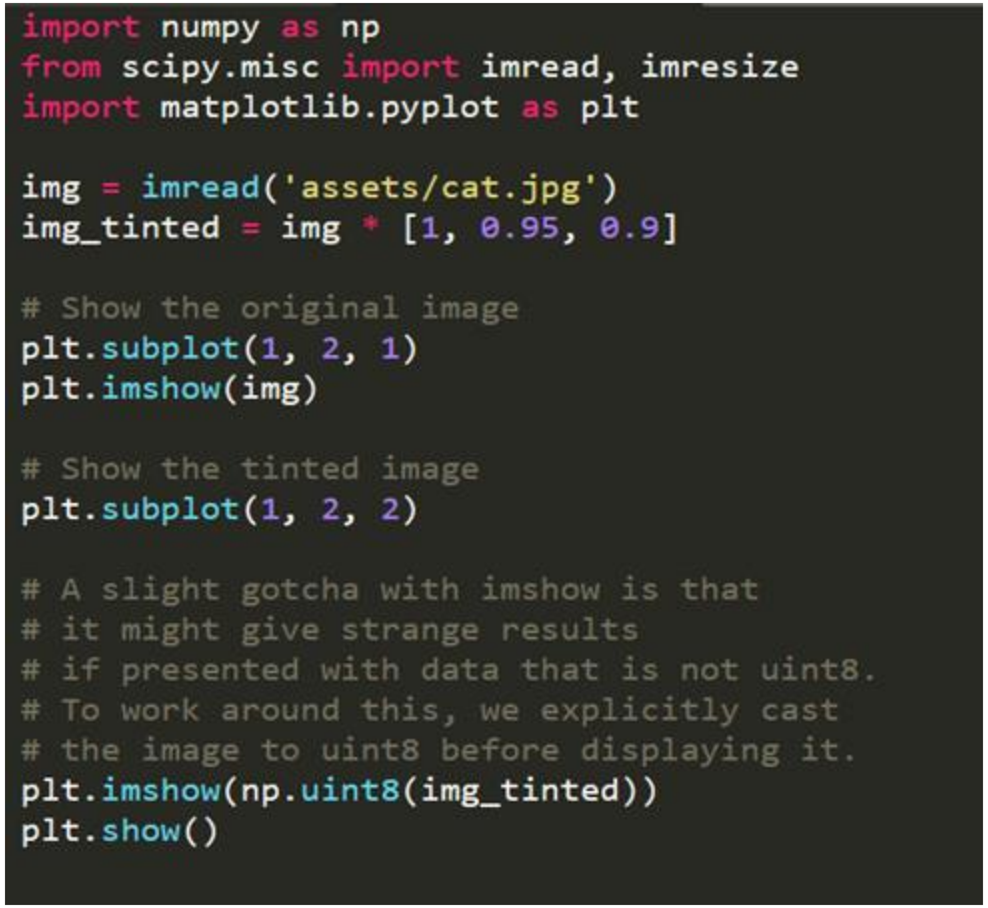
```python
import numpy as np
import matplotlib.pyplot as plt

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

# Set up a subplot grid that has height 2
# and width 1, and set the first such subplot
# as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make
# the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



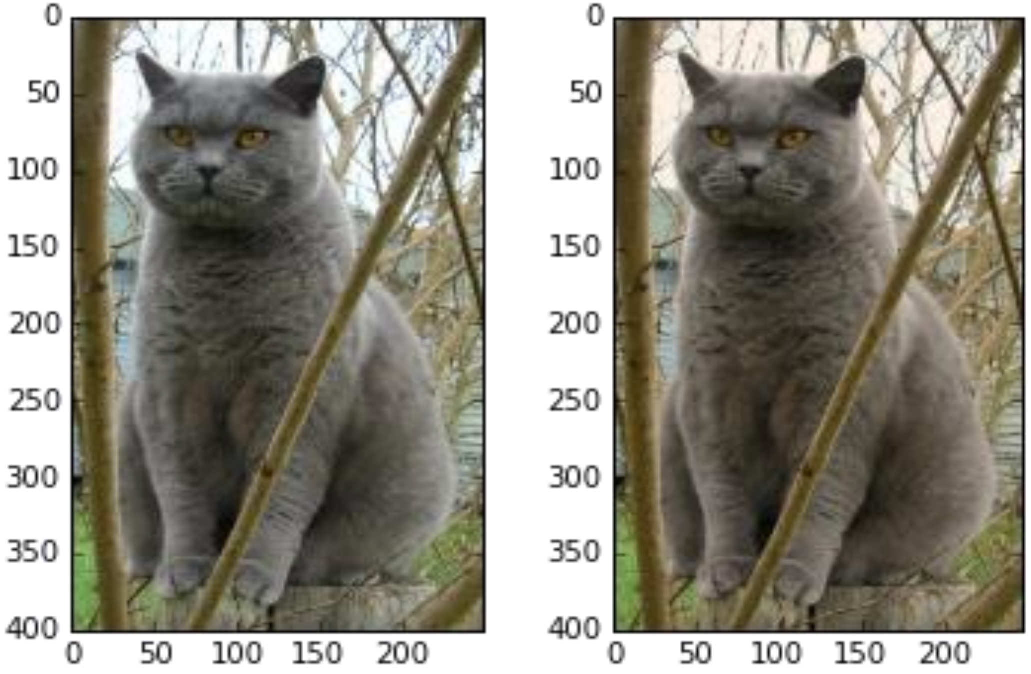
ref: https://cs231n.github.io/python-numpy-tutorial/

Displaying Images

- You can use the **imshow** function to show images.

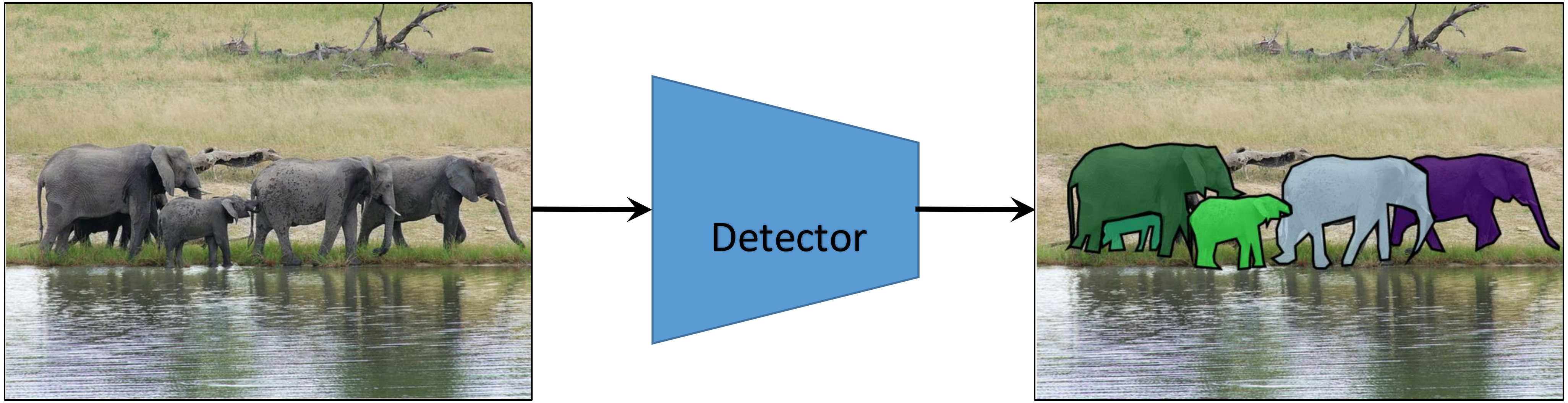


```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that
# it might give strange results
# if presented with data that is not uint8.
# To work around this, we explicitly cast
# the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```

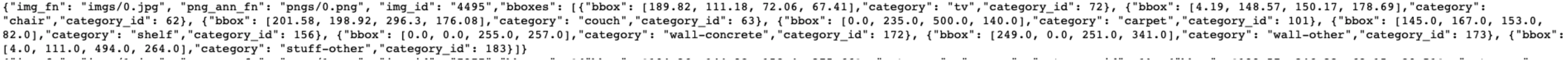


ref: https://cs231n.github.io/python-numpy-tutorial/

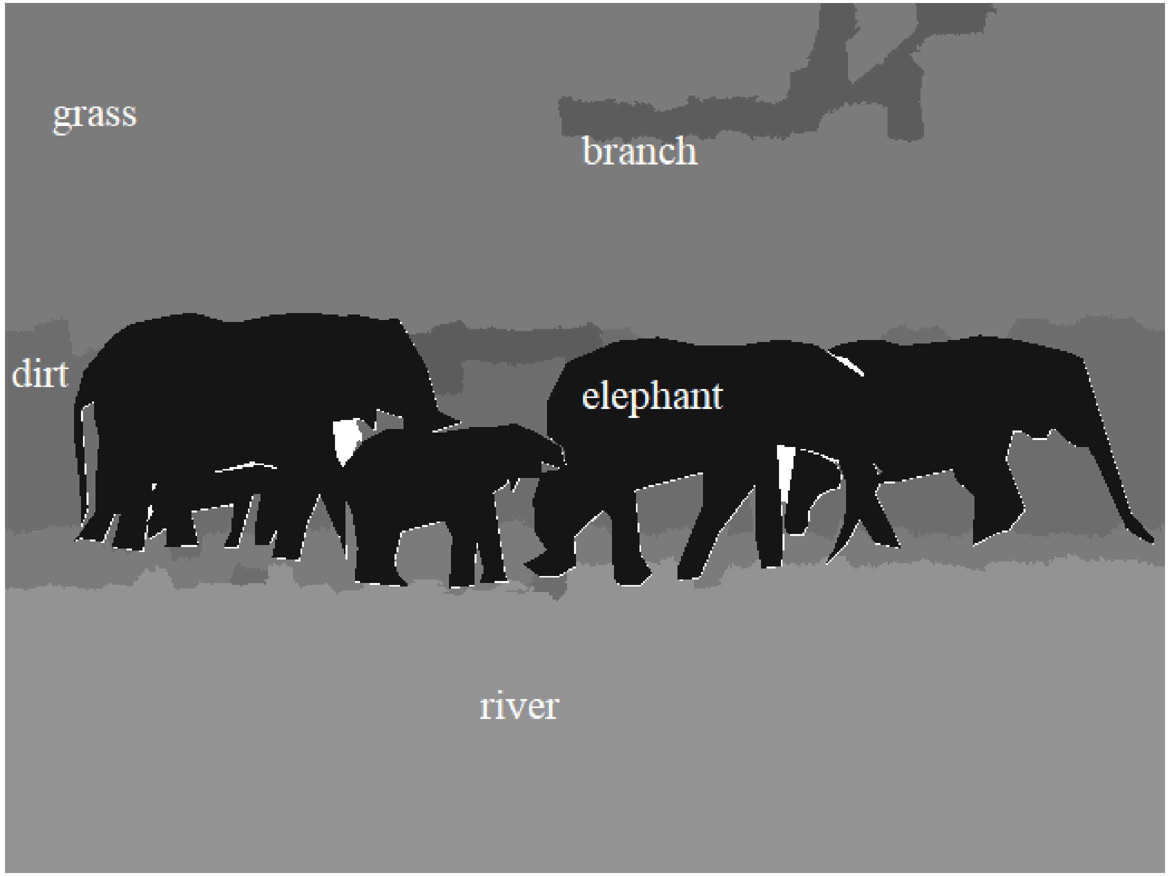
Object Segmentation



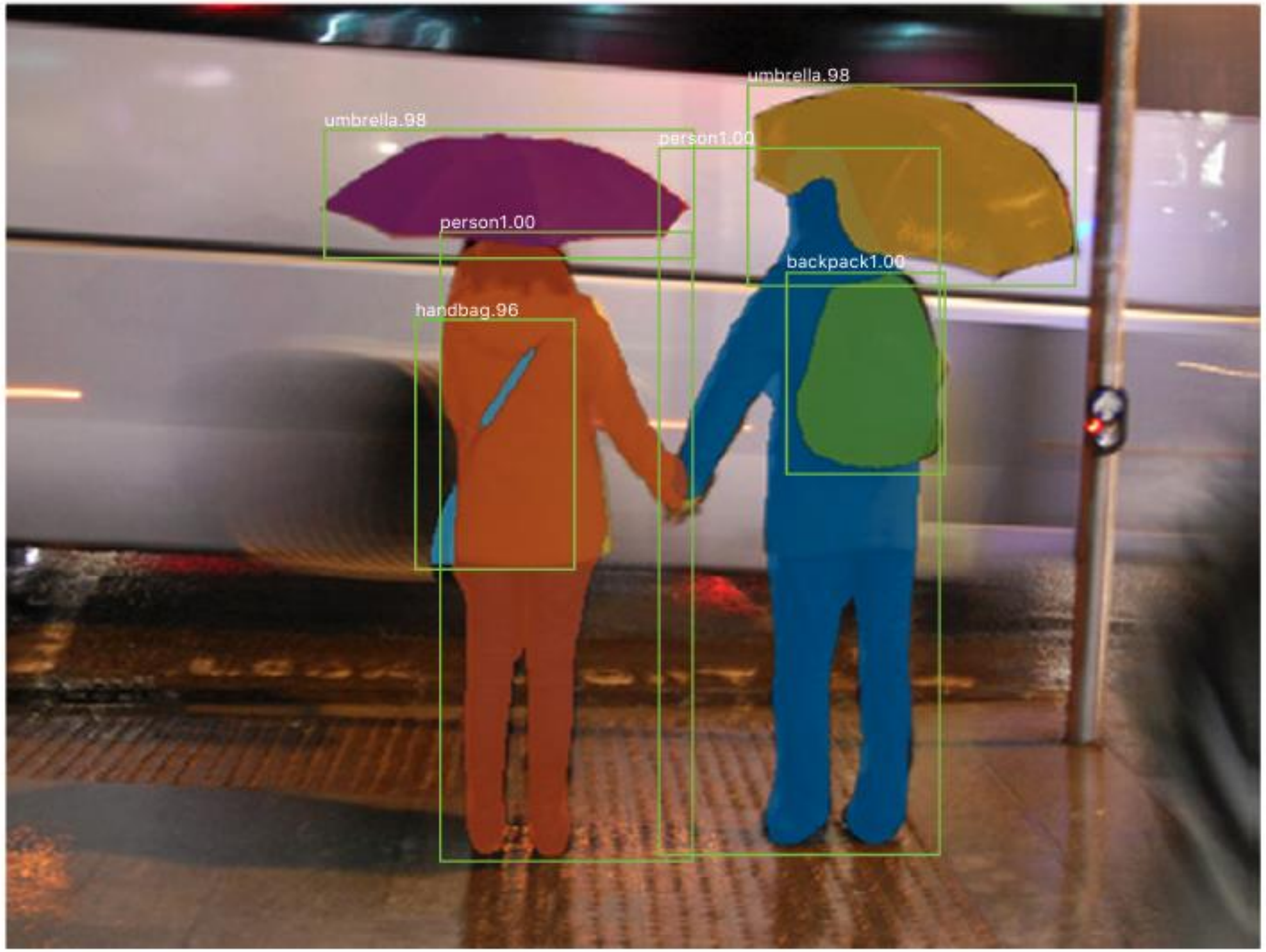
Object Segmentation/Detection – Annotation



```
{"img_fn": "imgs/0.jpg", "png_ann_fn": "pngs/0.png", "img_id": "4495","bboxes": [{"bbox": [189.82, 111.18, 72.06, 67.41],"category": "tv","category_id": 72}, {"bbox": [4.19, 148.57, 150.17, 178.69],"category":
"chair","category_id": 62}, {"bbox": [201.58, 198.92, 296.3, 176.08],"category": "couch","category_id": 63}, {"bbox": [0.0, 235.0, 500.0, 140.0],"category": "carpet","category_id": 101}, {"bbox": [145.0, 167.0, 153.0,
82.0],"category": "shelf","category_id": 156}, {"bbox": [0.0, 0.0, 255.0, 257.0],"category": "wall-concrete","category_id": 172}, {"bbox": [249.0, 0.0, 251.0, 341.0],"category": "wall-other","category_id": 173}, {"bbox":
[4.0, 111.0, 494.0, 264.0],"category": "stuff-other","category_id": 183}]}
```



Output


umbrella.98
umbrella.98
person1.00
person1.00
backpack1.00
handbag.96

Thank You