# cs224n-python-review-code-updated

January 15, 2021

### 0.0.1 Agenda

1. Installation
2. Basics
3. Iterables
4. Numpy (for math and matrix operations)
5. Matplotlib (for plotting)
6. Q&A

[1]:
```
# Note: This tutorial is based on Python 3.8
#       but it should apply to all Python 3.X versions
# Please note that this tutorial is NOT exhaustive
# We try to cover everything you need for class assignments
# but you should also navigate external resources
#
# More tutorials:
# NUMPY:
# https://cs231n.github.io/python-numpy-tutorial/#numpy
# https://numpy.org/doc/stable/user/quickstart.html
# MATPLOTLIB:
# https://matplotlib.org/gallery/index.html
# BASICS:
# https://www.w3schools.com/python/
# CONSULT THESE WISELY:
# The official documentation, Google, and Stack-overflow are your friends!
```

### 0.0.2 1. Installation

**Anaconda for environment management** https://www.anaconda.com/

common commands

conda env list <– list all environments

conda create -n newenv python=3.8 <– create new environment

conda enc create -f env.yml <– create environment from config file

conda activate envname <– activate a environment

conda deactivate <– exit environment

pip install packagename <– install package for current environment

jupyter notebook <– open jupyter in current environment

**Package installation using conda/pip** Live demo

**Recommended IDEs** Spyder (in-built in Anaconda)

Pycharm (the most popular choice, compatible with Anaconda)

[2]:
```
# common anaconda commands
#conda env list
#conda create -n name python=3.8
#conda env create -f env.yml
#conda activate python2.7
#conda deactivate
#install packages
#pip install <package>
```

### 0.0.3 2. Basics

https://www.w3schools.com/python/

[3]:
```
# input and output
name = input()
print("hello, " + name)
```

```
224N
hello, 224N
```

[4]:
```
# print multiple variables separated by a space
print("hello", name, 1, 3.0, True)
```

```
hello 224N 1 3.0 True
```

[5]:
```
# line comment
"""
block
comments
"""
```

[5]:
```
'\nblock \ncomments\n'
```

[6]:
```
# variables don't need explicit declaration
var = "hello" # string
var = 10.0    # float
var = 10      # int
```

```
var = True    # boolean
var = [1,2,3] # pointer to list
var = None    # empty pointer
```

[7]:
```
# type conversions
var = 10
print(int(var))
print(str(var))
print(float(var))
```

```
10
10
10.0
```

[8]:
```
# basic math operations
var = 10
print("var + 4 =", 10 + 4)
print("var - 4 =", 10 - 4)
print("var * 4 =", 10 * 4)
print("var ^ 4=", 10 ** 4)
print("int(var) / 4 =", 10//4)   # // for int division
print("float(var) / 4 =", 10/4)  # / for float division
# All compound assignment operators available
# including += -= *= **= /= //=
# pre/post in/decrementers not available (++ --)
```

```
var + 4 = 14
var - 4 = 6
var * 4 = 40
var ^ 4= 10000
int(var) / 4 = 2
float(var) / 4 = 2.5
```

[9]:
```
# basic boolean operations include "and", "or", "not"
print("not True is", not True)
print("True and False is", True and False)
print("True or False is", True or False)
```

```
not True is False
True and False is False
True or False is True
```

[10]:
```
# String operations
# '' and "" are equivalent
s = "String"
#s = 'Mary said "Hello" to John'
#s = "Mary said \"Hello\" to John"
```

```
# basic
print(len(s)) # get length of string and any iterable type
print(s[0]) # get char by index
print(s[1:3]) # [1,3)
print("This is a " + s + "!")

# handy tools
print(s.lower())
print(s*4)
print("ring" in s)
print(s.index("ring"))

# slice by delimiter
print("I am a sentence".split(" "))
# concatenate a list of string using a delimiter
print("...".join(['a','b','c']))

# formatting variables
print("Formatting a string like %.2f"%(0.12345))
print(f"Or like {s}!")
```

```
6
S
tr
This is a String!
string
StringStringStringString
True
2
['I', 'am', 'a', 'sentence']
a...b...c
Formatting a string like 0.12
Or like String!
```

[ ]:
```
# control flows
# NOTE: No parentheses or curly braces
#       Indentation is used to identify code blocks
#       So never ever mix spaces with tabs
for i in range(0,5):
    for j in range(i, 5):
        print("inner loop")
    print("outer loop")
```

[11]:
```
# if-else
var = 10
if var > 10:
```

```
    print(">")
elif var == 10:
    print("=")
else:
    print("<")
```

```
=
```

[12]:
```
# use "if" to check null pointer or empty arrays
var = None
if var:
    print(var)
var = []
if var:
    print(var)
var = "object"
if var:
    print(var)
```

```
object
```

[13]:
```
# while-loop
var = 5
while var > 0:
    print(var)
    var -=1
```

```
5
4
3
2
1
```

[14]:
```
# for-loop
for i in range(3):  # prints 0 1 2
    print(i)

"""
equivalent to
for (int i = 0; i < 3; i++)
"""
print("-------")
# range (start-inclusive, stop-exclusive, step)
for i in range(2, -3, -2):
    print(i)
"""
equivalent to
```

```
for (int i = 2; i > -3; i-=2)
"""
```

```
0
1
2
-------
2
0
-2
```

[14]:
```
'\nequivalent to\nfor (int i = 2; i > -3; i-=2)\n'
```

[15]:
```
# define function
def func(a, b):
    return a + b
func(1,3)
```

[15]:
```
4
```

[16]:
```
# use default parameters and pass values by parameter name
def rangeCheck(a, min_val = 0, max_val=10):
    return min_val < a < max_val    # syntactic sugar
rangeCheck(5, max_val=4)
```

[16]:
```
False
```

[17]:
```
# define class
class Foo:

    # optinal constructor
    def __init__(self, x):
        # first parameter "self" for instance reference, like "this" in JAVA
        self.x = x

    # instance method
    def printX(self): # instance reference is required for all function␣
 ↪parameters
        print(self.x)

    # class methods, most likely you will never need this
    @classmethod
    def printHello(self):
        print("hello")

obj = Foo(6)
obj.printX()
```

6

[18]:
```
# class inheritance - inherits variables and methods
# You might need this when you learn more PyTorch
class Bar(Foo):
    pass
obj = Bar(3)
obj.printX()
```

3

### 0.0.4 3. Iterables

[19]:
```
alist = list()  # linear, size not fixed, not hashable
atuple = tuple() # linear, fixed size, hashable
adict = dict()  # hash table, not hashable, stores (key,value) pairs
aset = set()    # hash table, like dict but only stores keys
acopy = alist.copy() # shallow copy
print(len(alist)) # gets size of any iterable type
```

0

[20]:
```
# examplar tuple usage
# creating a dictionary to store ngram counts
d = dict()
d[("a","cat")] = 10
d[["a","cat"]] = 11
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-47597361a541> in <module>
      3 d = dict()
      4 d[("a","cat")] = 10
----> 5 d[["a","cat"]] = 11

TypeError: unhashable type: 'list'
```

[21]:
```
"""
List: not hashable (i.e. can't use as dictionary key)
      dynamic size
      allows duplicates and inconsistent element types
      dynamic array implementation
"""
# list creation
alist = []          # empty list, equivalent to list()
alist = [1,2,3,4,5] # initialized list
```

```python
print(alist[0])
alist[0] = 5
print(alist)

print("-"*10)
# list indexing
print(alist[0]) # get first element (at index 0)
print(alist[-2]) # get last element (at index len-1)
print(alist[3:]) # get elements starting from index 3 (inclusive)
print(alist[:3]) # get elements stopping at index 3 (exclusive)
print(alist[2:4]) # get elements within index range [2,4)
print(alist[6:]) # prints nothing because index is out of range
print(alist[::-1]) # returns a reversed list

print("-"*10)
# list modification
alist.append("new item") # insert at end
alist.insert(0, "new item") # insert at index 0
alist.extend([2,3,4]) # concatenate lists
# above line is equivalent to alist += [2,3,4]
alist.index("new item") # search by content
alist.remove("new item") # remove by content
alist.pop(0) # remove by index
print(alist)

print("-"*10)
if "new item" in alist:
    print("found")
else:
    print("not found")

print("-"*10)
# list traversal
for ele in alist:
    print(ele)

print("-"*10)
# or traverse with index
for i, ele in enumerate(alist):
    print(i, ele)
```

```
1
[5, 2, 3, 4, 5]
----------
5
4
[4, 5]
```

```
[5, 2, 3]
[3, 4]
[]
[5, 4, 3, 2, 5]
----------
[2, 3, 4, 5, 'new item', 2, 3, 4]
----------
found
----------
2
3
4
5
new item
2
3
4
----------
0 2
1 3
2 4
3 5
4 new item
5 2
6 3
7 4
```

[22]:
```
"""
Tuple: hashable (i.e. can use as dictionary key)
       fixed size (no insertion or deletion)
"""
# it does not make sense to create empty tuples
atuple = (1,2,3,4,5)
 # or you can cast other iterables to tuple
atuple = tuple([1,2,3])

# indexing and traversal are same as list
```

[23]:
```
"""
Named tuples for readibility
"""
from collections import namedtuple
Point = namedtuple('Point', 'x y')
pt1 = Point(1.0, 5.0)
pt2 = Point(2.5, 1.5)
print(pt1.x, pt1.y)
```

```
1.0 5.0
```

[24]:
```
"""
Dict: not hashable
      dynamic size
      no duplicates allowed
      hash table implementation which is fast for searching
"""
# dict creation
adict = {} # empty dict, equivalent to dict()
adict = {'a':1, 'b':2, 'c':3}
print(adict)

# get all keys in dictionary
print(adict.keys())

# get value paired with key
print(adict['a'])
key = 'e'

# NOTE: accessing keys not in the dictionary leads to exception
if key in adict:
    print(adict[key])

# add or modify dictionary entries
adict['e'] = 10 # insert new key
adict['e'] = 5  # modify existing keys

print("-"*10)
# traverse keys only
for key in adict:
    print(key, adict[key])

print("-"*10)
# or traverse key-value pairs together
for key, value in adict.items():
    print(key, value)

print("-"*10)
# NOTE: Checking if a key exists
key = 'e'
if key in adict: # NO .keys() here please!
    print(adict[key])
else:
    print("Not found!")
```

```
{'a': 1, 'b': 2, 'c': 3}
dict_keys(['a', 'b', 'c'])
1
```

```
----------
a 1
b 2
c 3
e 5
----------
a 1
b 2
c 3
e 5
----------
5
```

[25]:
```
"""
Special dictionaries
"""
# set is a dictionary without values
aset = set()
aset.add('a')

# deduplication short-cut using set
alist = [1,2,3,3,3,4,3]
alist = list(set(alist))
print(alist)

# default_dictionary returns a value computed from a default function
#     for non-existent entries
from collections import defaultdict
adict = defaultdict(lambda: 'unknown')
adict['cat'] = 'feline'
print(adict['cat'])
print(adict['dog'])
```

```
[1, 2, 3, 4]
feline
unknown
```

[26]:
```
# counter is a dictionary with default value of 0
#     and provides handy iterable counting tools
from collections import Counter

# initialize and modify empty counter
counter1 = Counter()
counter1['t'] = 10
counter1['t'] += 1
counter1['e'] += 1
print(counter1)
```

```
print("-"*10)

# initialize counter from iterable
counter2 = Counter("letters to be counted")
print(counter2)
print("-"*10)

# computations using counters
print("1", counter1 + counter2)
print("2,", counter1 - counter2)
print("3", counter1 or counter2) # or for intersection, and for union
```

```
Counter({'t': 11, 'e': 1})
----------
Counter({'e': 4, 't': 4, ' ': 3, 'o': 2, 'l': 1, 'r': 1, 's': 1, 'b': 1, 'c': 1,
'u': 1, 'n': 1, 'd': 1})
----------
1 Counter({'t': 15, 'e': 5, ' ': 3, 'o': 2, 'l': 1, 'r': 1, 's': 1, 'b': 1, 'c':
1, 'u': 1, 'n': 1, 'd': 1})
2, Counter({'t': 7})
3 Counter({'t': 11, 'e': 1})
```

[27]:
```
# sorting
a = [4,6,1,7,0,5,1,8,9]
a = sorted(a)
print(a)
a = sorted(a, reverse=True)
print(a)
```

```
[0, 1, 1, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 1, 1, 0]
```

[28]:
```
# sorting
a = [("cat",1), ("dog", 3), ("bird", 2)]
a = sorted(a)
print(a)
a = sorted(a, key=lambda x:x[1])
print(a)
```

```
[('bird', 2), ('cat', 1), ('dog', 3)]
[('cat', 1), ('bird', 2), ('dog', 3)]
```

[29]:
```
# useful in dictionary sorting
adict = {'cat':3, 'bird':1}
print(sorted(adict.items(), key=lambda x:x[1]))
```

```
[('bird', 1), ('cat', 3)]
```

[30]:
```
# Syntax sugar: one-line control flow + list operation
sent = ["i am good", "a beautiful day", "HELLO FRIEND"]
"""
for i in range(len(sent)):
    sent[i] = sent[i].lower().split(" ")
"""
sent1 = [s.lower().split(" ") for s in sent]
print(sent1)

sent2 = [s.lower().split(" ") for s in sent if len(s) > 10]
print(sent2)

# Use this for deep copy!
# copy = [obj.copy() for obj in original]
```

```
[['i', 'am', 'good'], ['a', 'beautiful', 'day'], ['hello', 'friend']]
[['a', 'beautiful', 'day'], ['hello', 'friend']]
```

[31]:
```
# Syntax sugar: * operator for repeating iterable elements
print("-"*10)
print([1]*10)

# Note: This only repeating by value
#       So you cannot apply the trick on reference types

# To create a double list
# DONT
doublelist = [[]]*10
doublelist[0].append(1)
print(doublelist)
# DO
doublelist = [[] for _ in range(10)]
doublelist[0].append(1)
print(doublelist)
```

```
----------
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
[[1], [], [], [], [], [], [], [], [], []]
```

### 0.0.5 4. Numpy

Very powerful python tool for handling matrices and higher dimensional arrays

[32]:
```
import numpy as np
```

[33]:
```
# create arrays
a = np.array([[1,2],[3,4],[5,6]])
```

```
print(a)
print(a.shape)
# create all-zero/one arrays
b = np.ones((3,4)) # np.zeros((3,4))
print(b)
print(b.shape)
# create identity matrix
c = np.eye(5)
print(c)
print(c.shape)
```

```
[[1 2]
 [3 4]
 [5 6]]
(3, 2)
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
(3, 4)
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
(5, 5)
```

[34]:
```
# reshaping arrays
a = np.arange(8)          # [8,] similar range() you use in for-loops
b = a.reshape((4,2))      # shape [4,2]
c = a.reshape((2,2,-1))   # shape [2,2,2] -- -1 for auto-fill
d = c.flatten()           # shape [8,]
e = np.expand_dims(a, 0) # [1,8]
f = np.expand_dims(a, 1) # [8,1]
g = e.squeeze()           # shape[8, ]    -- remove all unnecessary dimensions
print(a)
print(b)
```

```
[0 1 2 3 4 5 6 7]
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

[35]:
```
# concatenating arrays
a = np.ones((4,3))
b = np.ones((4,3))
c = np.concatenate([a,b], 0)
```

```
print(c.shape)
d = np.concatenate([a,b], 1)
print(d.shape)
```

```
(8, 3)
(4, 6)
```

[36]:
```
# one application is to create a batch for NN
x1 = np.ones((32,32,3))
x2 = np.ones((32,32,3))
x3 = np.ones((32,32,3))
# --> to create a batch of shape (3,32,32,3)
x = [x1, x2, x3]
x = [np.expand_dims(xx, 0) for xx in x] # xx shape becomes (1,32,32,3)
x = np.concatenate(x, 0)
print(x.shape)
```

```
(3, 32, 32, 3)
```

[37]:
```
# access array slices by index
a = np.zeros([10, 10])
a[:3] = 1
a[:, :3] = 2
a[:3, :3] = 3
rows = [4,6,7]
cols = [9,3,5]
a[rows, cols] = 4
print(a)
```

```
[[3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 4.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 4. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 4. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]]
```

[38]:
```
# transposition
a = np.arange(24).reshape(2,3,4)
print(a.shape)
print(a)
a = np.transpose(a, (2,1,0)) # swap 0th and 2nd axes
print(a.shape)
print(a)
```

```
(2, 3, 4)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
(4, 3, 2)
[[[ 0 12]
  [ 4 16]
  [ 8 20]]

 [[ 1 13]
  [ 5 17]
  [ 9 21]]

 [[ 2 14]
  [ 6 18]
  [10 22]]

 [[ 3 15]
  [ 7 19]
  [11 23]]]
```

[39]:
```
c = np.array([[1,2],[3,4]])
# pinv is pseudo inversion for stability
print(np.linalg.pinv(c))
# l2 norm by default, read documentation for more options
print(np.linalg.norm(c))
# summing a matrix
print(np.sum(c))
# the optional axis parameter
print(c)
print(np.sum(c, axis=0)) # sum along axis 0
print(np.sum(c, axis=1)) # sum along axis 1
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
5.477225575051661
10
[[1 2]
 [3 4]]
[4 6]
[3 7]
```

[40]:
```
# dot product
c = np.array([1,2])
d = np.array([3,4])
print(np.dot(c,d))
```

```
11
```

[41]:
```
# matrix multiplication
a = np.ones((4,3)) # 4,3
b = np.ones((3,2)) # 3,2 --> 4,2
print(a @ b)       # same as a.dot(b)
c = a @ b          # (4,2)

# automatic repetition along axis
d = np.array([1,2,3,4]).reshape(4,1)
print(c + d)
# handy for batch operation
batch = np.ones((3,32))
weight = np.ones((32,10))
bias = np.ones((1,10))
print((batch @ weight + bias).shape)
```

```
[[3. 3.]
 [3. 3.]
 [3. 3.]
 [3. 3.]]
[[4. 4.]
 [5. 5.]
 [6. 6.]
 [7. 7.]]
(3, 10)
```

[42]:
```
# speed test: numpy vs list
a = np.ones((100,100))
b = np.ones((100,100))

def matrix_multiplication(X, Y):
    result = [[0]*len(Y[0]) for _ in range(len(X))]
    for i in range(len(X)):
        for j in range(len(Y[0])):
            for k in range(len(Y)):
                result[i][j] += X[i][k] * Y[k][j]
    return result

import time

# run numpy matrix multiplication for 10 times
```

```
start = time.time()
for _ in range(10):
    a @ b
end = time.time()
print("numpy spends {} seconds".format(end-start))

# run list matrix multiplication for 10 times
start = time.time()
for _ in range(10):
    matrix_multiplication(a,b)
end = time.time()
print("list operation spends {} seconds".format(end-start))

# the difference gets more significant as matrices grow in size!
```

```
numpy spends 0.001990079879760742 seconds
list operation spends 8.681961059570312 seconds
```

[43]:
```
# element-wise operations, for examples
np.log(a)
np.exp(a)
np.sin(a)
# operation with scalar is interpreted as element-wise
a * 3
```

[43]:
```
array([[3., 3., 3., …, 3., 3., 3.],
       [3., 3., 3., …, 3., 3., 3.],
       [3., 3., 3., …, 3., 3., 3.],
       …,
       [3., 3., 3., …, 3., 3., 3.],
       [3., 3., 3., …, 3., 3., 3.],
       [3., 3., 3., …, 3., 3., 3.]])
```

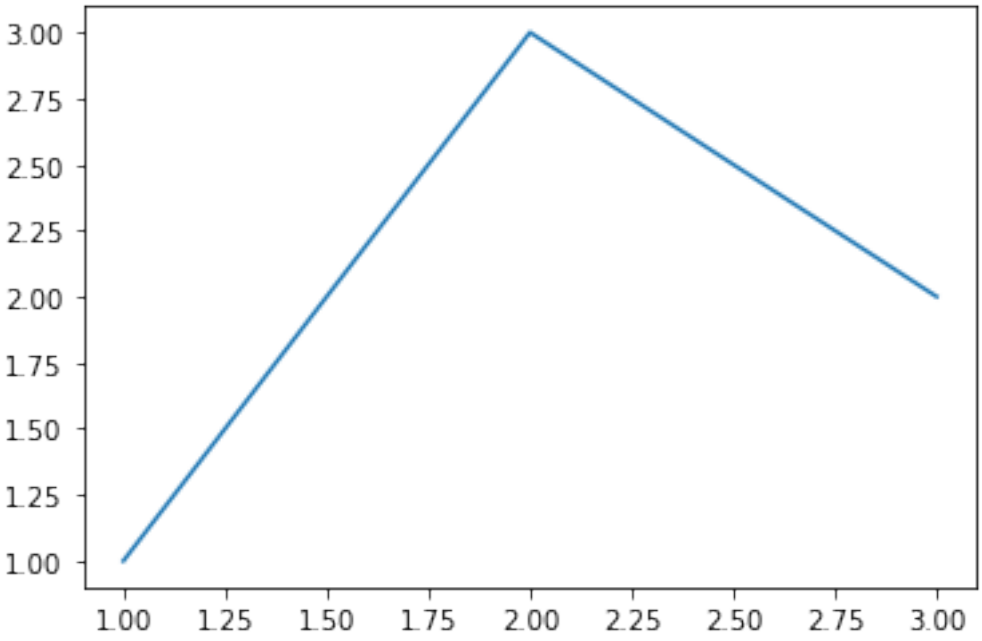
### 0.0.6 5. Matplotlib

Powerful tool for visualization Many tutorials online. We only go over the basics here

[44]:
```
import matplotlib.pyplot as plt
```

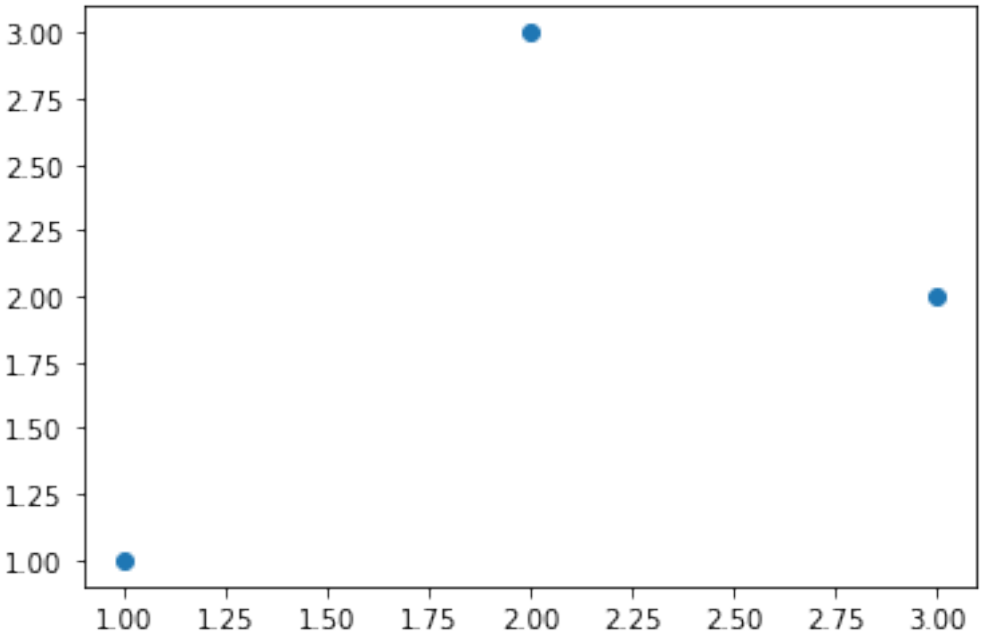
[45]:
```
# line plot
x = [1,2,3]
y = [1,3,2]
plt.plot(x,y)
```

[45]:
```
[<matplotlib.lines.Line2D at 0x17b1b50a040>]
```



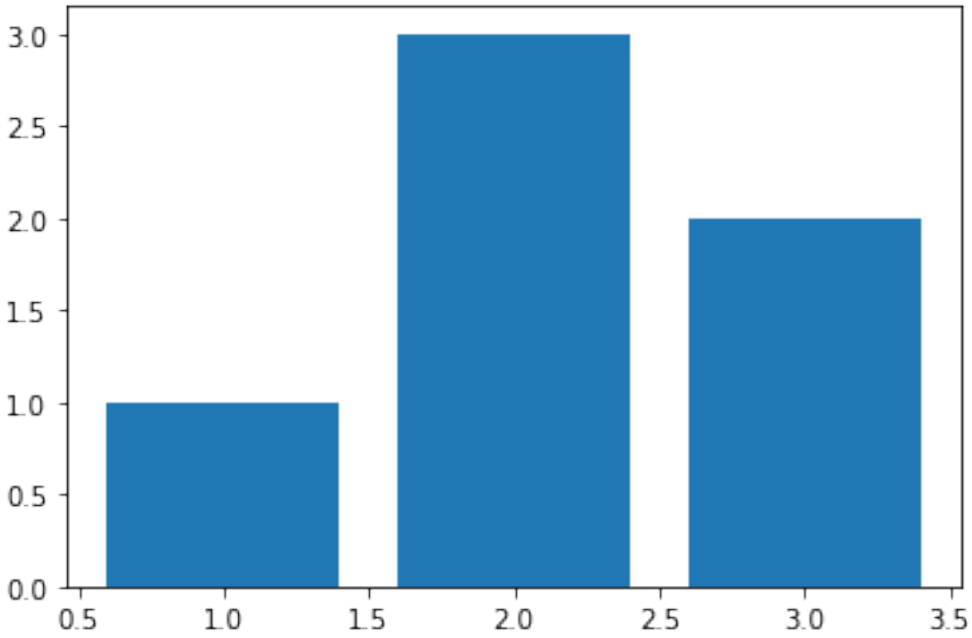

[46]:
```
# scatter plot
plt.scatter(x,y)
```

[46]: <matplotlib.collections.PathCollection at 0x17b1b530490>



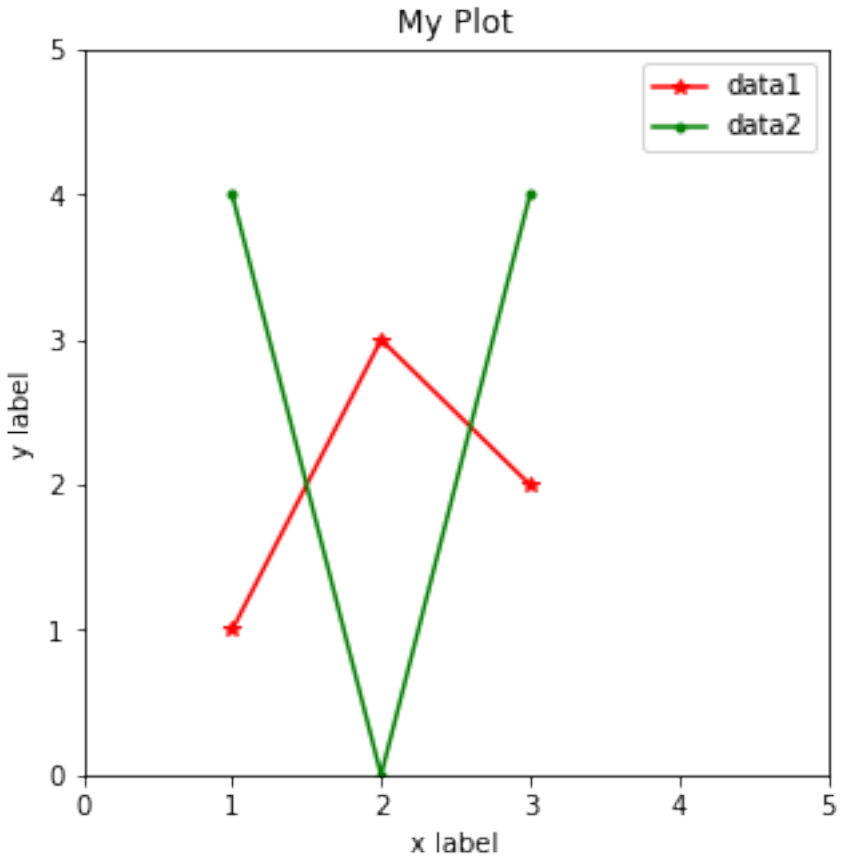
[47]:
```
# bar plots
plt.bar(x,y)
```

[47]:
```
<BarContainer object of 3 artists>
```




[48]:
```
# plot configurations
x = [1,2,3]
y1 = [1,3,2]
y2 = [4,0,4]

# set figure size
plt.figure(figsize=(5,5))

# set axes
plt.xlim(0,5)
plt.ylim(0,5)
plt.xlabel("x label")
plt.ylabel("y label")

# add title
plt.title("My Plot")

plt.plot(x,y1, label="data1", color="red", marker="*")
plt.plot(x,y2, label="data2", color="green", marker=".")
plt.legend()
```

[48]: <matplotlib.legend.Legend at 0x17b1b669d00>




### 0.0.7 Q&A

[ ]: