Basic Python Tutorial (Python3)

Edited By: Jagabandhu Mishra, IIT Dharwad

Date: 5/06/2020

Acknowledgement:

Volodymyr Kuleshov and Isaac Caswell from the `CS231n` Python tutorial by Justin Johnson (http://cs231n.github.io/python-numpy-tutorial/).

Introduction

Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

We expect that many of you will have some experience with Python and numpy; for the rest of you, this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

Some of you may have previous knowledge in Matlab, in which case we also recommend the numpy for Matlab users page (https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html).

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Images
- Dataframe: Panda
- Pytorch

Tools used to exicute python scripts:

1) Linux terminal

2) Spyder (like MATLAB editor)

3) Jupyter Lab/ notebook (for documentation, like MATLAB live editor (.mlx file))

Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

In [ ]:
```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
```

```
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

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

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

Small analysis of re-cursive function calling

In [ ]:
```
arr=[9,8,7,6,5,4,3,2,1]
pivot=arr[len(arr)//2]
print(pivot)
left = [x for x in arr if x < pivot]
print(left)
middle = [x for x in arr if x == pivot]
print(middle)
right = [x for x in arr if x > pivot]
print(right)
```

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

In [ ]:
```
quicksort(left)
```

Out[ ]:
```
[1, 2, 3, 4]
```

In [ ]:
```
quicksort(right)
```

Out[ ]:
```
[6, 7, 8, 9]
```

Python versions

There are currently two different supported versions of Python, 2 and 3. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2 may not work under 3 and vice versa. For this class all code will use Python 3.

You can check your Python version at the command line by running `python --version` .

Basic data types

Numbers

Integers and floats work as you would expect from other languages:

In [ ]:
```
x = 3
print(x), type(x)
```

```
3
```

Out[ ]:
```
(None, int)
```

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

```
4
2
6
9
```

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

```
4
8
```

In [ ]:
```
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
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation.

Booleans

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&& , || , etc.):

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

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

Now we let's look at the operations:

In [ ]:
```
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
```

Strings

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

```
hello 5
```

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

```
hello world
```

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

```
hello world 12
```

String objects have a bunch of useful methods; for example:

In [1]:
```
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())      # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints
```

```
print(s.center(7))     # Center a string, padding with spaces; prints " hello
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with
                                # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints "w
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

You can find a list of all string methods in the documentation.

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists

A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

In [2]:
```
xs = [3, 1, 2]   # Create a list
print(xs, xs[2])
print(xs[-1])     # Negative indices count from the end of the list; prints '
```

```
[3, 1, 2] 2
2
```

In [ ]:
```
xs[2] = 'foo'    # Lists can contain elements of different types
print(xs)
```

```
[3, 1, 'foo']
```

In [ ]:
```
xs.append('bar') # Add a new element to the end of the list
print(xs)
```

```
[3, 1, 'foo', 'bar']
```

In [ ]:
```
x = xs.pop()     # Remove and return the last element of the list
print(x, xs)
```

```
foo [3, 1]
```

As usual, you can find all the gory details about lists in the documentation.

Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

In [ ]:
```
nums = list(range(5))    # range is a built-in function that creates a list o
print(nums)          # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])     # Get a slice from index 2 to 4 (exclusive); prints "[2,
print(nums[2:])      # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])      # Get a slice from the start to index 2 (exclusive); prin
print(nums[:])       # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
print(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(nums)          # Prints "[0, 1, 8, 9, 4]"
```

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

```
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

Loops

You can loop over the elements of a list like this:

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

```
cat
dog
monkey
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

In [ ]:
```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

List comprehensions:

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

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

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

You can make this code simpler using a list comprehension:

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

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

List comprehensions can also contain conditions:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

```
[0, 4, 16]
```

Dictionaries

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

In [ ]:
```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['dog'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
```

```
furry
True
```

In [ ]:
```
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
```

```
wet
```

In [ ]:
```
print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-78fc9745d9cf> in <module>()
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

NameError: name 'd' is not defined
```

In [ ]:
```
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-21-c19d3fcbe97a> in <module>()
----> 1 print(d.get('monkey', 'N/A'))  # Get an element with a default; print
s "N/A"
      2 print(d.get('fish', 'N/A'))    # Get an element with a default; print
s "wet"

NameError: name 'd' is not defined
```

In [ ]:
```
del(d['fish'])        # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

You can find all you need to know about dictionaries in the documentation.

It is easy to iterate over the keys in a dictionary:

In [ ]:
```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

If you want access to keys and their corresponding values, use the iteritems method:

In [ ]:
```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

In [ ]:
```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
```

```
True
False
```

In [ ]:
```
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

```
True
3
```

In [ ]:
```
animals.add('cat')       # Adding an element that is already in the set does
print(len(animals))
animals.remove('cat')    # Remove an element from a set
print(len(animals))
```

```
3
2
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

In [ ]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints("#1: fish", "#2: dog", "#3: cat")
```

```
#1: dog
#2: cat
#3: fish
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

In [ ]:
```
from math import sqrt
print({int(x) for x in range(30)})
print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
1, 22, 23, 24, 25, 26, 27, 28, 29}
{0, 1, 2, 3, 4, 5}
```

Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

In [ ]:
```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)       # Create a tuple
print(type(t))
print(d)
print(d[t])
print(d[(1, 2)])
```

```
<class 'tuple'>
{(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5, (6, 7): 6,
(7, 8): 7, (8, 9): 8, (9, 10): 9}
5
1
```

Functions

Python functions are defined using the `def` keyword. For example:

In [ ]:
```
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
```

We will often define functions to take optional keyword arguments, like this:

In [ ]:
```
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
```

Classes

The syntax for defining classes in Python is straightforward:

In [ ]:
```
class Greeter:

    # Constructor
    def __init__(self, name):
        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)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

```
Hello, Fred
HELLO, FRED!
```

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. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

To use Numpy, we first need to import the `numpy` package:

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

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.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

In [ ]:
```
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]
```

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

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

In [ ]:
```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

In [ ]:
```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)
```

```
[[0. 0.]
 [0. 0.]]
```

In [ ]:
```
b = np.ones((1,2))   # Create an array of all ones
print(b)
```

```
[[1. 1.]]
```

In [ ]:
```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

In [ ]:
```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

In [ ]:
```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

```
[[0.16416542 0.94484031]
 [0.22839339 0.96542329]]
```

Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

In [ ]:
```
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]])
print(a)
# 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):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[2 3]
 [6 7]]
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

In [ ]:
```
print(a[0, 1])
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

In [ ]:
```
# 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]])
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Two ways of accessing the data in the middle row of the array. Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:

In [ ]:
```
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # 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]] (1, 4)
[[5 6 7 8]] (1, 4)
```

In [ ]:
```
# 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)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

In [ ]:
```
a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
[1 4 5]
```

In [ ]:
```
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

In [ ]:
```
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

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

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[ 1  6  7 11]
```

In [ ]:
```
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

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

```
a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

In [ ]:

```
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

In [ ]:

```
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

You can read all about numpy datatypes in the documentation.

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

In [ ]:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

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

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

In [ ]:

```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
```

```
[[-4. -4.]
 [-4. -4.]]
```

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

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

In [ ]:
```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

In [ ]:
```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, `*` is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

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

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

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

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

```
[29 67]
[29 67]
```

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

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

In [ ]:
```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

In [ ]:
```
print(x)
print(x.T)
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

In [ ]:
```
v = np.array([[1,2,3]])
print(v)
print(v.T)
```

```
[[1 2 3]]
[[1]
 [2]
 [3]]
```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

In [ ]:
```
# 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 = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

In [ ]:
```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

In [ ]:
```
y = x + vv  # Add x and vv elementwise
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

In [ ]:
```
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)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

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.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

In [ ]:
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

In [ ]:
```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

In [ ]:
```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

In [ ]:
```
# Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

In [ ]:
```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

In [ ]:
```
# demo block (multiplication, power, dimension extension)

import numpy as np
l=np.array([1,2,3])
m=np.array([2,3,4])
m=np.expand_dims(m,axis=1)
l=np.expand_dims(l,axis=1)
l=l.T
prod=m*l
```

```
print(l.shape)
print(prod.shape)
print(prod)
print(np.power(prod,2))
sm=np.sum(prod,axis=1)# w.r.t row [1 2 3]=6
print(sm)
```

```
(1, 3)
(3, 3)
[[ 2  4  6]
 [ 3  6  9]
 [ 4  8 12]]
[[  4  16  36]
 [  9  36  81]
 [ 16  64 144]]
[12 18 24]
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

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

By running this special iPython command, we will be displaying plots inline:

In [ ]:
```
%matplotlib inline
```

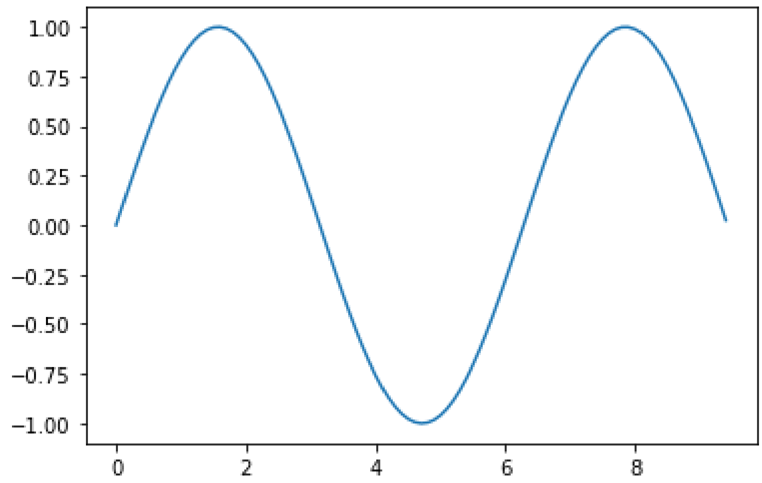
Plotting

The most important function in `matplotlib` is plot, which allows you to plot 2D data. Here is a simple example:

In [ ]:
```
# 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)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7fddc18746a0>]
```

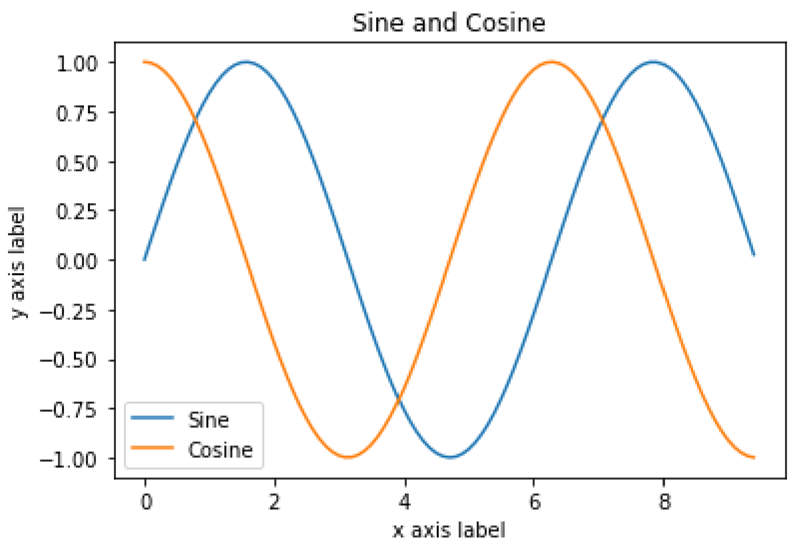


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

In [ ]:
```
y_sin = np.sin(x)
y_cos = np.cos(x)

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

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fddc0372c88>
```



Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

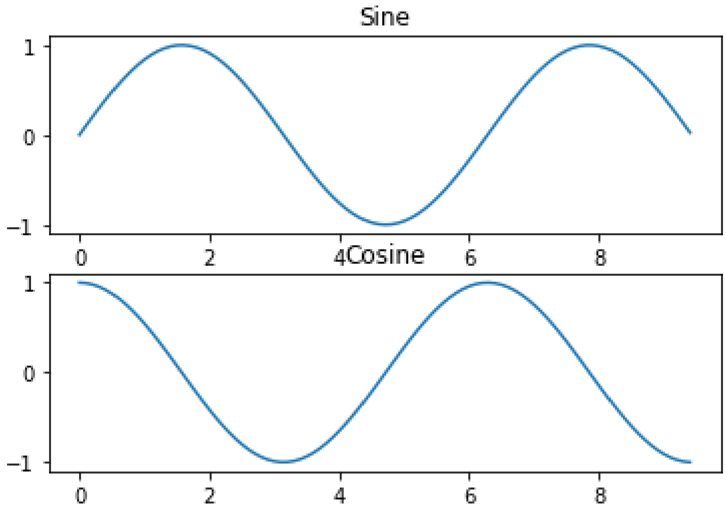
In [ ]:
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# 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()
```



You can read much more about the `subplot` function in the documentation.

Panda Dataframe:

Useful for importing data from csv files

In this notebook, we will introduce you to DataFrame from Pandas in python. You have already used it in one of the previous labs. However, this will give you a more specific material. As usual we import the packages.

In [ ]:
```
!pip3 install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packag
es (1.1.5)
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.7/dist
-packages (from pandas) (1.19.5)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/pytho
n3.7/dist-packages (from pandas) (2.8.1)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-
packages (from pandas) (2018.9)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-pack
ages (from python-dateutil>=2.7.3->pandas) (1.15.0)
```

In [ ]:
```
!conda install pandas
```

```
/bin/bash: conda: command not found
```

In [2]:
```
import numpy as np
import pandas as pd
```

Data Structures in Padas

Series

"one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.)" - https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html)

It's like table data type in MATLAB

Lets see the difference between Numpy and Panda

Define and Print `myNumpyArray' and` myPandaSeries' and note the differences between them.

In [ ]:
```
myNumpyArray = np.random.randn(4)
myPandaSeries = pd.Series(np.random.randn(4))

print(myNumpyArray)
print(myPandaSeries)
```

```
[ 1.44485231  0.24090179 -0.95759879  1.05472698]
0   -0.439734
1   -1.586188
2    0.281620
3   -0.565094
dtype: float64
```

In the below, generate 4 random numbers and custom label them as a, b c and d (see https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html)

In [ ]:
```
mySeries = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
print(mySeries)
```

```
a   -1.020684
b    2.202988
c   -0.664031
d   -0.365681
dtype: float64
```

Now, print the value at index `b`, in the following.

In [ ]:
```
mySeries['b']              # Write code to print the value at index b.
```

Out[ ]:
```
2.2029881464887016
```

What are the indexes of `mySeries` ?

In [ ]:
```
mySeries.index
```

Out[ ]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

Now, consider the following native Python dictionary.

In [ ]:
```
myDictionary = {'a': 1, 'b': 0, 'c': 2}
print(myDictionary)
```

```
{'a': 1, 'b': 0, 'c': 2}
```

Here, a, b and c are the dictionary indicies and corresponding values are 1 0 and 2. Now, let us convert these dictionary values to Pandas Series.

In [ ]:
```
pd.Series(myDictionary)
```

We can do mathematical operations using Series as well. Let us construct 2 Series and do some operations.

In [ ]:
```
mySeries1 = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
mySeries2 = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
mySeriesSum = mySeries1 + mySeries1            # Add the series
mySeriesInnerProduct =   mySeries1.T * mySeries1     # Find mySeries1.T*mySe
mySeriesExp1 = np.exp(mySeries1)               # Find the exponential of     my
```

DataFrame

DataFrame is like a excel sheet or Google sheet, where different columns can have different datatypes.

In [ ]:
```
d = {'First Col': [1., 2., 3., 4.], 'Second Column': [4., 3., 2., 1.]}
pd.DataFrame(d)
```

Out[ ]:

	First Col	Second Column
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

Now construct the DataFrame to give the following output (https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html):

```
               House Area (in Sq. Feet)   House Prize (in Lakh
   Rs)
         House-1             1000                      11
         House-2             1256                      13
         House-3             1530                      17
```

In [ ]:
```
d = {'House Area': pd.Series([1., 2., 3.], index=['House-1', 'House-2', 'Hous
   ....:      'House-2': pd.Series([1., 2., 3.], index=['House-1', 'House-2',
pd.DataFrame(d)
```

Out[ ]:

	House Area	House-2
House-1	1.0	1.0
House-2	2.0	2.0
House-3	3.0	3.0

Based on Practical Data

Now, import the .csv file which is already stored in Google Drive as a DataFrame. We have already downloaded the stock price from https://in.finance.yahoo.com/quote/BR/history?period1=1443830400&period2=1601683200&interval=1d&filter=history&frequency=1d for 5 years.

Download CSV file:
https://drive.google.com/file/d/1o8WjxKxnWCLNJ1pbW_sKPKZjlcRoYmE0/view?usp=sharing

In [3]:
```
stockpriceDF = pd.read_csv("/home/jagabandhu/Documents/IEEE_Summer_School/Br
```

It is often important to look at only some of the data available in practice, just to get a feel for the data. Try `df.head()` in the below.

In [4]:
```
stockpriceDF.head()
```

Out[4]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-10-05	57.029999	57.860001	56.990002	57.369999	52.438675	968400
1	2015-10-06	57.250000	57.470001	56.459999	56.619999	51.753147	475900
2	2015-10-07	56.820000	57.000000	56.200001	56.470001	51.616039	874800
3	2015-10-08	56.439999	57.279999	55.689999	57.049999	52.146183	495000
4	2015-10-09	57.070000	57.400002	56.990002	57.330002	52.402122	712700

Display the `Open` price for all the days.

In [5]:
```
stockpriceDF['Open']     # Enter your code here
# See
print(stockpriceDF['Open'])
```

```
0        57.029999
1        57.250000
2        56.820000
3        56.439999
4        57.070000
           ...    
1254    131.800003
1255    132.100006
1256    131.919998
1257    133.410004
1258    131.699997
Name: Open, Length: 1259, dtype: float64
```

There are many other operations which you can do. However, as an basic introductory one, we can stop here. You may take look at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html for further reference.

Pytorch

High level python library, like numpy, generally used for deep learning applications

Compared with NumPy arrays, PyTorch tensors have added advantage that both tensors and related operations can run on the CPU or GPU.

The second important thing that PyTorch provides allows tensors to keep track of the operations performed on them that helps to compute gradients or derivatives of an output with respect to any of its inputs.

Installation: https://anaconda.org/pytorch/pytorch

In [3]:
```
import torch
```

At its core, PyTorch is a library for processing tensors. A tensor is a number, vector, matrix, or any n-dimensional array. Let's create a tensor with a single number.

In [9]:
```
# Number
t1 = torch.tensor(4.)
t1
```

Out[9]:
```
tensor(4.)
```

In [5]:
```
# Vector
t2 = torch.tensor([1., 2, 3, 4])
t2
```

Out[5]:
```
tensor([1., 2., 3., 4.])
```

In [6]:
```
# Matrix
t3 = torch.tensor([[5., 6],
                   [7, 8],
                   [9, 10]])
t3
```

Out[6]:
```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
```

In [7]:
```
# 3-dimensional array
t4 = torch.tensor([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])
t4
```

Out[7]:
```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
```

In [11]:
```
print(t2.shape)
print(t1.dtype)
```

```
torch.Size([4])
torch.float32
```

We can combine tensors with the usual arithmetic operations. Let's look at an example:

In [12]:
```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
x, w, b
```

Out[12]:
```
(tensor(3.), tensor(4., requires_grad=True), tensor(5., requires_grad=True))
```

We've created three tensors: x, w, and b, all numbers. w and b have an additional parameter requires_grad set to True. We'll see what it does in just a moment.

Let's create a new tensor y by combining these tensors.

In [16]:
```
y = w * x + b
y
```

Out[16]:
```
tensor(17., grad_fn=<AddBackward0>)
```

As expected, y is a tensor with the value 3 * 4 + 5 = 17. What makes PyTorch unique is that we can automatically compute the derivative of y w.r.t. the tensors that have requires_grad set to True i.e. w and b. This feature of PyTorch is called autograd (automatic gradients).

To compute the derivatives, we can invoke the .backward method on our result y.

In [18]:
```
y
```

Out[18]:
```
tensor(17., grad_fn=<AddBackward0>)
```

In [17]:
```
y.backward()
```

In [19]:
```
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dx: None
dy/dw: tensor(6.)
dy/db: tensor(3.)
```

Tensor functions

Apart from arithmetic operations, the torch module also contains many functions for creating and manipulating tensors. Let's look at some examples.

In [ ]:
```
# Create a tensor with a fixed value for every element
t6 = torch.full((3, 2), 42)
t6
```

In [ ]:
```
# Concatenate two tensors with compatible shapes
t7 = torch.cat((t3, t6))
t7
```

In [ ]:
```
# Compute the sin of each element
t8 = torch.sin(t7)
t8
```

In [ ]:
```
# Change the shape of a tensor
t9 = t8.reshape(3, 2, 2)
t9
```

You can learn more about tensor operations here: https://pytorch.org/docs/stable/torch.html

Interoperability with Numpy

Numpy is a popular open-source library used for mathematical and scientific computing in Python. It enables efficient operations on large multi-dimensional arrays and has a vast ecosystem of supporting libraries, including:

Pandas for file I/O and data analysis

Matplotlib for plotting and visualization

OpenCV for image and video processing

In [23]:
```
import numpy as np

x = np.array([[1, 2], [3, 4.]])
y = torch.from_numpy(x) # numpy to torch
y
```

Out[23]:
```
tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

In [24]:
```
x.dtype, y.dtype
```

Out[24]:
```
(dtype('float64'), torch.float64)
```

In [26]:
```
z=y.numpy() # torch to numpy
z
```

Out[26]:
```
array([[1., 2.],
       [3., 4.]])
```

In [21]:
```
t1 = t1.to(dtype=torch.float64) # type casting
t1.dtype
```

Out[21]:
```
torch.float64
```

Thank you!, hope this will help you