HelloWorld

October 15, 2018

1 ECE 285 Assignment #1

2 Python, Numpy and Matplotlib

3 Mehrnaz Motamed

4 A53245061

---

5 1. Getting started – Python, Platform and Jupyter

```
In [160]: print ('HelloWorld!')
```

```
HelloWorld!
```

6 2. Numpy

6.1 2.1. Arrays

6.1.1 1. Run the following:

```
In [161]: import numpy as np
          a = np.array([1, 2, 3])
          print(type(a))
          print(a.shape)
          print(a[0], a[1], a[2])
          a[0] = 5
```

```
<type 'numpy.ndarray'>
(3,)
(1, 2, 3)
```

6.1.2 What are the rank, shape of a, and the current values of a[0], a[1], a[2]?

a = [1 2 3] type(a): <class 'numpy.ndarray'> a.shape: (3,) a[0], a[1], a[2]: 1 2 3 a[0] = 5 : a = [5, 2, 3] a is a 3 by 1 array, it has 2 dimensions, and its ***rank is 1. Shape of a is (3,).*** *The current values of a[0], a[1], a[2], are* ***5, 2, 3.***

6.1.3 2. Run the following:

```
In [162]: b = np.array([[1, 2, 3], [4, 5, 6]])

In [163]: b

Out[163]: array([[1, 2, 3],
                 [4, 5, 6]])
```

6.1.4 What are the rank, shape of b, and the current values of b[0, 0], b[0, 1], b[1, 0]?

```
In [164]: print(b.shape)
          print('b[0,0] is %d' %(b[0,0]))
          print('b[0,1] is %d' %(b[0,1]))
          print('b[1,0] is %d' %(b[1,0]))

(2, 3)
b[0,0] is 1
b[0,1] is 2
b[1,0] is 4
```

b is a 2 dimension array. In other words ***rank of b is 2****.* ***Shape of b is (2,3).*** *This means that b is 2 by 3.* ***Current values*** *of b[0,0], b[0,1], b[1,0] are* ***1, 2, 4.***

6.1.5 3. Numpy also provides many functions to create arrays. Assign the correct comment to each of the following instructions:

```
In [165]: a = np.zeros((2,2))
          b = np.ones((1,2))
          c = np.full((2,2), 7)
          d = np.eye(2)
          e = np.random.random((2,2))

In [166]: print(a)

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

a: #an array of all zeros

```
In [167]: print(b)
```

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

b: # an array of all ones

```
In [168]: print(c)

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

c: #a constant array

```
In [169]: print(d)

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

d: #a 2x2 identity matrix

```
In [170]: print(e)

[[0.68058582 0.97548079]
 [0.34193945 0.13452789]]
```

e: # an array filled with random values

6.2 2.2 Array indexing

6.2.1 4. Run the following

```
In [171]: a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
          b = a[:2, 1:3]

In [172]: print(a)

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


In [173]: print(b)

[[2 3]
 [6 7]]
```

6.2.2 What are the shape of a and b? What are the values in b?

```
In [174]: print(a.shape)
```

```
(3, 4)
```

```
In [175]: print(b.shape)
```

```
(2, 2)
```

a is 3 by 4 and b is 2 by 2. The values in b are 2,3,6,7.

6.2.3 5. A slice of an array is a view into the same data. Follow the comments in the following snippet

```
In [176]: print(a[0, 1]) # Prints "2"
          b[0, 0] = 77
          print(a[0, 1])
```

```
2
77
```

6.2.4 What does the last line print? Does modifying b modify the original array?

*The last line prints **77**. This is because **modifying b modifies the original array** which is a. In other words, changing b[0,0] results in changing a[0,1].*

6.2.5 6. 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. Create the following rank 2 array with shape (3, 4):

```
In [177]: a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

```
In [178]: print(a)
```

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

```
In [179]: print(a.shape)
```

```
(3, 4)
```

6.2.6 Run the following:

```
In [180]: 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
          print(row_r1, row_r1.shape) # Prints "[5 6 7 8] (4,)"
          print(row_r2, row_r2.shape) # Prints "[[5 6 7 8]] (1, 4)"
          col_r1 = a[:, 1]
          col_r2 = a[:, 1:2]
```

```
(array([5, 6, 7, 8]), (4,))
(array([[5, 6, 7, 8]]), (1, 4))
```

row_r1 is the second row of a with rank 1. row_r1 is array([5, 6, 7, 8]) with the shape of (4,). row_r2 is the same with rank 2. row_r2 is array([[5, 6, 7, 8]]), (1, 4).

6.2.7 What are the values and shapes of col r1 and col r2?

```
In [181]: print(col_r1)
```

```
[ 2  6 10]
```

```
In [182]: print(col_r2)
```

```
[[ 2]
 [ 6]
 [10]]
```

```
In [183]: print(col_r1.shape)
```

```
(3,)
```

```
In [184]: print(col_r2.shape)
```

```
(3, 1)
```

Values of col_r1 are [2 6 10] with the shape (3,) while values of col_r2 are [[2]; [6]; [10]] with the shape (3,1). In other words col_r1 has rank 1 while col_r2 has rank 2.

6.2.8 7. Run the following:

```
In [185]: a = np.array([[1,2], [3, 4], [5, 6]])
          print(a[[0, 1, 2], [0, 1, 0]])
          print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

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

6.2.9 Are the two printed arrays equivalent?

```
In [186]: print(a)

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


In [187]: print(a[[0, 1, 2], [0, 1, 0]].shape)

(3,)


In [188]: print(np.array([a[0, 0], a[1, 1], a[2, 0]]).shape)

(3,)
```

Yes, they are equivalent.

6.2.10 8. When using integer array indexing, you can duplicate several times the same element from the source array. Compare the following instructions:

```
In [189]: b = a[[0, 0], [1, 1]]
          c = np.array([a[0, 1], a[0, 1]])

In [190]: print(b)

[2 2]


In [191]: print(c)

[2 2]


In [192]: b.shape

Out[192]: (2,)

In [193]: c.shape

Out[193]: (2,)
```

Both b and c store the element a[0,1] twice. In b, we first indicate the row numbers and then the column numbers while in c we indicate row and column of each element and then row and column of the next element and so forth.

6.2.11 9. One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix. Run the following code.

```
In [194]: a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
          b = np.array([0, 2, 0, 1])
          print(a[np.arange(4), b])
          a[np.arange(4), b] += 10
          print(a)
```

```
[ 1  6  7 11]
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

6.2.12 What do you observe?

```
In [195]: 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 [196]: b = np.array([0, 2, 0, 1])
          print(b)
```

```
[0 2 0 1]
```

```
In [197]: print(a[np.arange(4), b])
```

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

```
In [198]: a[np.arange(4), b] += 10
```

```
In [199]: print(a)
```

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

np.arrange(4) gives values [0, 1, 2, 3]. Then a.[np.arrange(4),b] is b = a[[0, 1, 2, 3], [0, 2, 0, 1]] which is [1, 6, 7, 11]. Then we add 10 to all of these values which gives us a new a that is [[11 2 3] [4 5 16] [17 8 9] [10 21 12]].

6.2.13 10. Run the following:

```
In [200]: a = np.array([[1,2], [3, 4], [5, 6]])
          bool_idx = (a > 2)

In [201]: print(a)

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


In [202]: print(bool_idx)

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


In [203]: a.shape

Out[203]: (3, 2)

In [204]: bool_idx.shape

Out[204]: (3, 2)
```

6.2.14 What is the result stored in bool_idx?

This will give us an array with the same shape and dimensions as a. In this matrix, each element is True if the boolean is true and false if the condition is satisfied.

6.2.15 Run the following:

```
In [205]: print(a[bool_idx])
          print(a[a > 2])

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

6.2.16 What are the values printed? What is their shape?

```
In [206]: a[bool_idx].shape

Out[206]: (4,)

In [207]: a[a > 2].shape

Out[207]: (4,)
```

Both return an array of (4,) of the elements that satisfy the condition which is $a > 2$ which is also described in bool_idx.

6.3 2.3 Datatypes

6.3.1 1. Here is an example:

```
In [208]: x = np.array([1, 2]) # Let numpy choose the datatype
          print(x.dtype)
          y = np.array([1.0, 2.0]) # Let numpy choose the datatype
          print(y.dtype)
          z = np.array([1, 2], dtype=np.int32) # Force a particular datatype
          print(z.dtype)
```

```
int64
float64
int32
```

6.3.2 What are the datatypes of x, y, z?

```
In [209]: print(x)
```

```
[1 2]
```

```
In [210]: print(y)
```

```
[1. 2.]
```

```
In [211]: print(z)
```

```
[1 2]
```

*Data type of x is **64 bit integer**, while data type of y is **64 bit float**, and type of z as specified is **32 bit integer.***

6.4 2.4 Array math

6.4.1 12. Give the output for each print statement in the following code snippet

```
In [212]: 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))
          # Elementwise difference; both produce the array
          print(x - y)
          print(np.subtract(x, y))
          # Elementwise product; both produce the array
          print(x * y)
          print(np.multiply(x, y))
```

```
          # Elementwise division; both produce the array
          print(x / y)
          print(np.divide(x, y))
          # Elementwise square root; produces the array
          print(np.sqrt(x))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

```
In [213]: x = np.array([[1,2],[3,4]], dtype=np.float64)
          print(x)
```

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

```
In [214]: y = np.array([[5,6],[7,8]], dtype=np.float64)
          # Elementwise sum; both produce the array
          print(y)
```

```
[[5. 6.]
 [7. 8.]]
```

```
In [215]: print(x + y)
```

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

```
In [216]: print(np.add(x, y))
```

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

x+y and np.add(x, y) give the same result which is adding each element of x to the corresponding element in y.

```
In [217]: print(x - y)

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


In [218]: print(np.subtract(x, y))

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

x-y and np.subtract(x, y) give the same result which is subtracting each element of y from the corresponding element in x.

```
In [219]: print(x * y)

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


In [220]: print(np.multiply(x, y))

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

*x*y and np.multiply(x, y) give the same result which is multiplying each element of y to the corresponding element in x.*

```
In [221]: print(x / y)

[[0.2        0.33333333]
 [0.42857143 0.5       ]]


In [222]: print(np.divide(x, y))

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

x/y and np.divide(x, y) give the same result which is dividing each element of x by the corresponding element in y.

```
In [223]: print(np.sqrt(x))

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

np.sqrt(x) gives the square root of each element of x.

```
In [224]: print(x*x)

[[ 1.  4.]
 [ 9. 16.]]
```

*We can see that multiplication and division and square root are defined elementwise.

6.4.2 13. What are the mathematical operations performed by the last two instructions?

```
In [225]: v = np.array([9,10])
          w = np.array([11, 12])
          print(v.dot(w))
          print(np.dot(v, w))

219
219


In [226]: print(v)

[ 9 10]


In [227]: print(w)

[11 12]
```

The dot product is defined as $w^T v$ or $v^T w$. In other words it is the summation of product of corresponding elements. It can be shown both as v.dot(w) and np.dot(v, w).

6.4.3 14. What are the mathematical operations performed in the snippet below?

```
In [228]: x = np.array([[1,2],[3,4]])
          print(x.dot(v))
          print(np.dot(y, v))

[29 67]
[105. 143.]
```

```
In [229]: print(x)

[[1 2]
 [3 4]]


In [230]: print(v)

[ 9 10]


In [231]: print(x.dot(v))

[29 67]


In [232]: print(np.dot(x,v))

[29 67]


In [233]: print(np.dot(v,x))

[39 58]
```

x.dot(v) returns the dot product of each row of x defined by [] with v.

```
In [234]: print(y)

[[5. 6.]
 [7. 8.]]


In [235]: print(v)

[ 9 10]


In [236]: print(np.dot(y, v))

[105. 143.]
```

np.dot(y, v) returns the dot product of each row of y defined by [] with v.

```
In [237]: print(np.dot(v,y))

[115. 134.]
```

While np.dot(v,y) returns the dot product of each column of y with v

6.4.4 15. Write a code to compute the product of the matrices x with the following matrix y

```
In [238]: y = np.array([[5,6,7],[7,8,9]])

In [239]: print(y)

[[5 6 7]
 [7 8 9]]


In [240]: print(x)

[[1 2]
 [3 4]]


In [241]: print(np.dot(x,y))

[[19 22 25]
 [43 50 57]]
```

6.4.5 Can you write the same code for the product of y with x?

```
In [242]: print(np.dot(y,x))



        ValueErrorTraceback (most recent call last)

        <ipython-input-242-54a09557b32d> in <module>()
    ----> 1 print(np.dot(y,x))


        ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

It can be seen that np.dot(y,x) gives an error because the dimensions of y and x do not agree for the multiplication in the order of yx.

```
In [243]: print(np.dot(np.transpose(y),x))

[[26 38]
 [30 44]
 [34 50]]
```

Because y is 2 by 3 and x is 2 by 2, only x.y has meaning and we cannot multiply them in the y.x order unless we transpose y first.

```
In [244]: 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]


In [245]: print(x)

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

6.4.6 16. In the following:

```
In [246]: x = np.array([[1,2,3], [3,4,5]])
          print(x)
          print(x.T)

[[1 2 3]
 [3 4 5]]
[[1 3]
 [2 4]
 [3 5]]


In [247]: print(x)

[[1 2 3]
 [3 4 5]]


In [248]: print(x.T)

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

6.4.7 What is the transpose of x? How is it different from x?

Transpose of x or x.T is a matrix resulted from changing rows and columns of the original matrix x.

```
In [249]: x.T.shape

Out[249]: (3, 2)
```

```
In [250]: x.shape

Out[250]: (2, 3)

In [251]: type(x)

Out[251]: numpy.ndarray

In [252]: type(x.T)

Out[252]: numpy.ndarray
```

6.5 2.5 Broadcasting

6.5.1 17. Complete the following code in order to add the vector v to each row of a matrix x:

```
In [253]: x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
          v = np.array([1,0,1])
          y = np.zeros(x.shape)
          for i in range(x.shape[0]):
              for j in range(x.shape[1]):
                  y[i,j] = x[i,j] + v[j]

In [254]: print(y)

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


In [255]: print(np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]]))

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


In [256]: print(np.array([1,0,1]))

[1 0 1]


In [257]: print(np.zeros(x.shape))

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

```
In [258]: print(x.shape[0])
```

```
4
```

```
In [259]: print(x.shape[1])
```

```
3
```

6.5.2 18. The previous solution works well; however when the matrix x is very large, computing an explicit loop in Python could be slow. An alternative is to use tile. Run the following and interpret the result:

```
In [260]: vv = np.tile(v, (4, 1))
          print(vv)
```

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

vv = np.tile(v, (4,1)) generates a matrix of rank 2 in which v is repeated 4 by 1 times. In other words, vv is repeating v in 4 rows and one column.

6.5.3 Next, complete the code to compute y from x and vv without explicit loops.

```
In [261]: y = x +vv
          print(y)
```

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

6.5.4 19. Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. This is simply obtained as follows:

```
In [262]: 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
```

```
In [263]: print(x)
```

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

```
In [264]: print(v)

[1 0 1]


In [265]: print(x+v)

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

6.5.5 Compute the outer product of v and w using broadcasting:

```
In [266]: v = np.array([1,2,3]) # v has shape (3,)
          w = np.array([4,5]) # w has shape (2,)

In [267]: print(v)

[1 2 3]


In [268]: print(w)

[4 5]


In [269]: print(np.reshape(v, (3, 1)))

[[1]
 [2]
 [3]]


In [270]: print(np.dot(np.reshape(v,(3,1)),np.reshape(w,(1,2))))

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


In [271]: w.shape

Out[271]: (2,)

In [272]: v.shape

Out[272]: (3,)

In [273]: np.reshape(v, (3, 1)).shape
```

```
Out[273]: (3, 1)

In [274]: print(np.dot(np.reshape(v, (3, 1)), np.reshape(w, (1, 2))))

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

6.5.6 20. Add v to each row of x using broadcasting:

```
In [275]: x = np.array([[1,2,3], [4,5,6]])

In [276]: print(x)

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


In [277]: print(v)

[1 2 3]


In [278]: print(np.reshape(v,(1,3))+np.reshape(x,(2,3)))

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

6.5.7 21. Add w to each column of x using broadcasting.

```
In [279]: print(w)

[4 5]


In [280]: print(np.reshape(w,(2,1))+np.reshape(x,(2,3)))

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

6.5.8 22. Numpy treats scalars as arrays of shape (). Write a code to multiply x with 2.

```
In [281]: print(2*x)

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

```
In [282]: print(x*2)

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


In [283]: print(np.full((2,1),2)*x)

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


In [284]: print([[2],[2]]*x)

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

6.5.9 2.6 Numpy Documentation

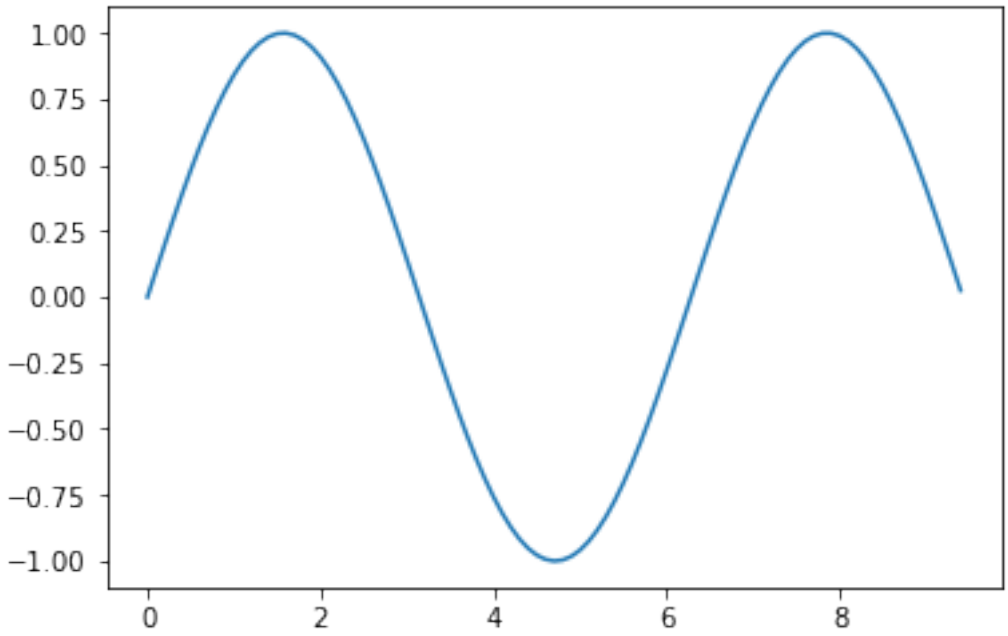
6.6 3. Matplotlib

6.6.1 3.1 Plotting

6.6.2 An important function in matplotlib is plot, which allows you to plot 2D graphs. Here is a simple example:

6.6.3 23. Run the example and check that it produces the following plot:

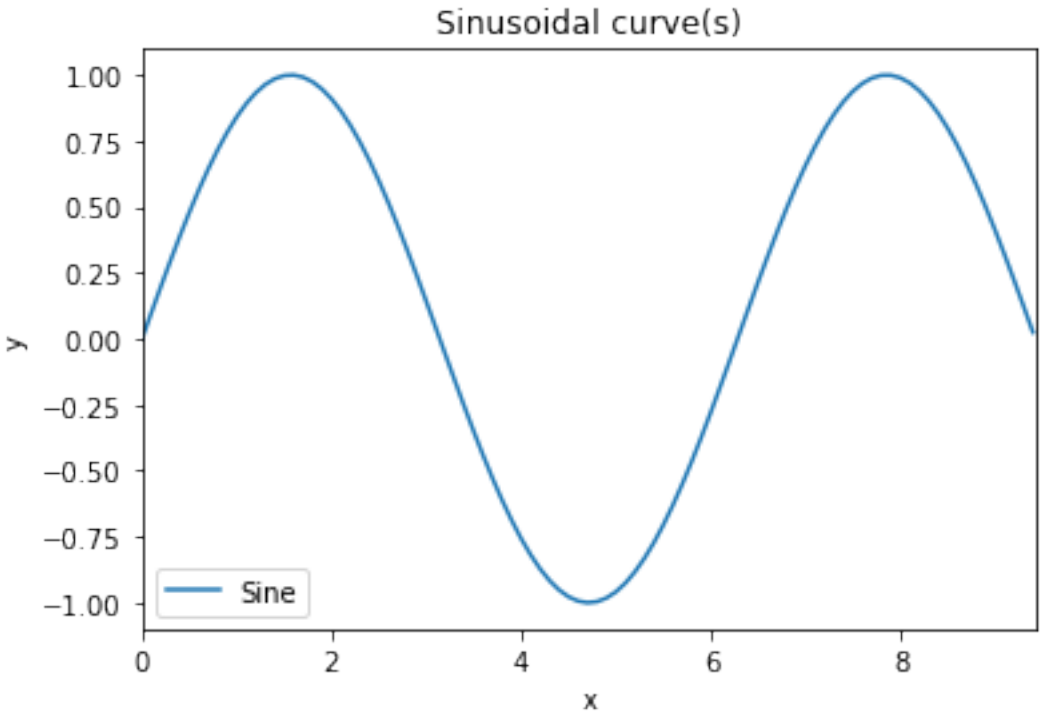
```
In [285]: import numpy as np
          import matplotlib.pyplot as plt


          # 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.
```



6.6.4 24. Modify the above example by adding these extra instructions before plt.show()

In [286]:
```
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)



#############################This is the added section
plt.xlim(0, 3 * np.pi)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sinusoidal curve(s)')
plt.legend(['Sine'])
#####################################################
```

```
plt.show() # You must call plt.show() to make graphics appear.
```

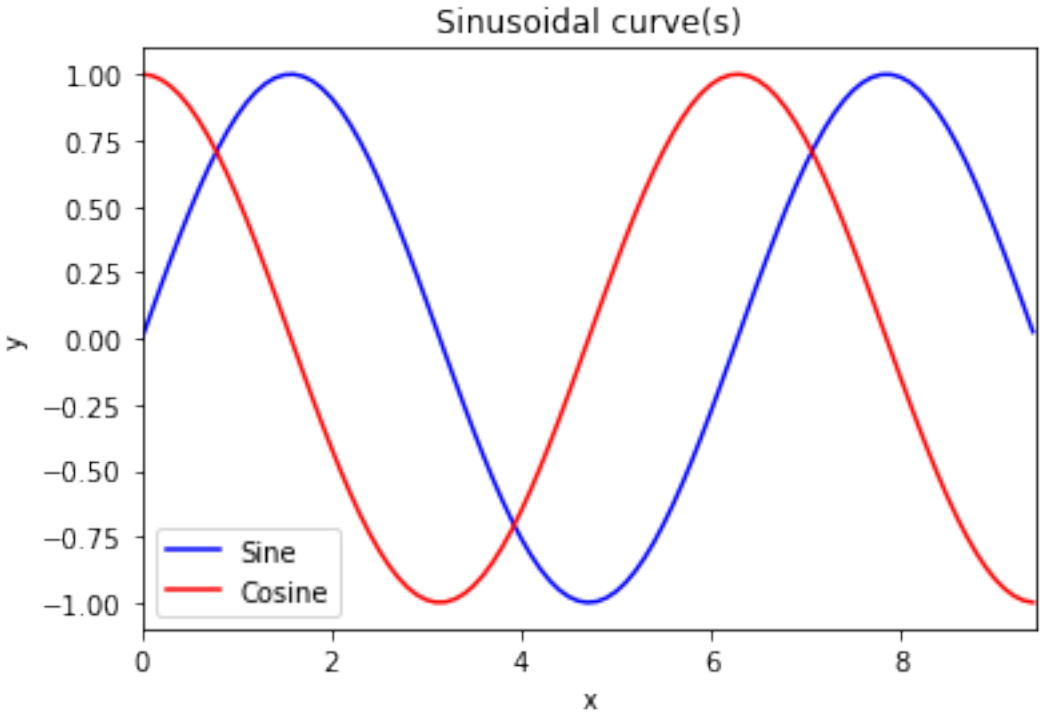


plt.xlim(0, 3 np.pi) shows the plot only in range of (0,3pi). plt.xlabel('x') labels x axis as 'x'. plt.ylabel('y') labels y axis as 'y'. plt.title('Sinusoidal curve(s)') determines the title of the plot as 'Sinusoidal curve(s)'. plt.legend(['Sine']) labels the specific function as 'Sine' (blue line) to avoid confusions when plotting several functions at the same time.*

6.6.5 25. Write a code to plot both sine and cosine curves in the same plot with proper legend.

```
In [287]: 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)
          z = np.cos(x)


          # Plot the points using matplotlib
          plt.plot(x,y,'b')
          plt.plot(x,z,'r')

          plt.xlim(0, 3 * np.pi)
```

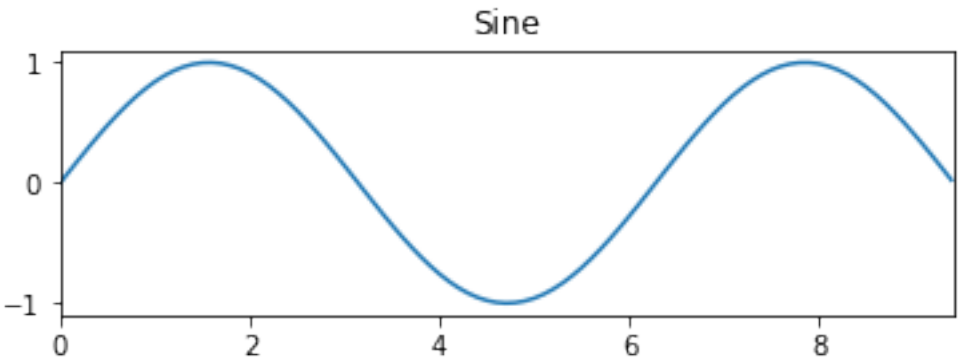
```
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sinusoidal curve(s)')
plt.legend(['Sine','Cosine'])

plt.show() # You must call plt.show() to make graphics appear.
```



6.7 3.2 Subplots

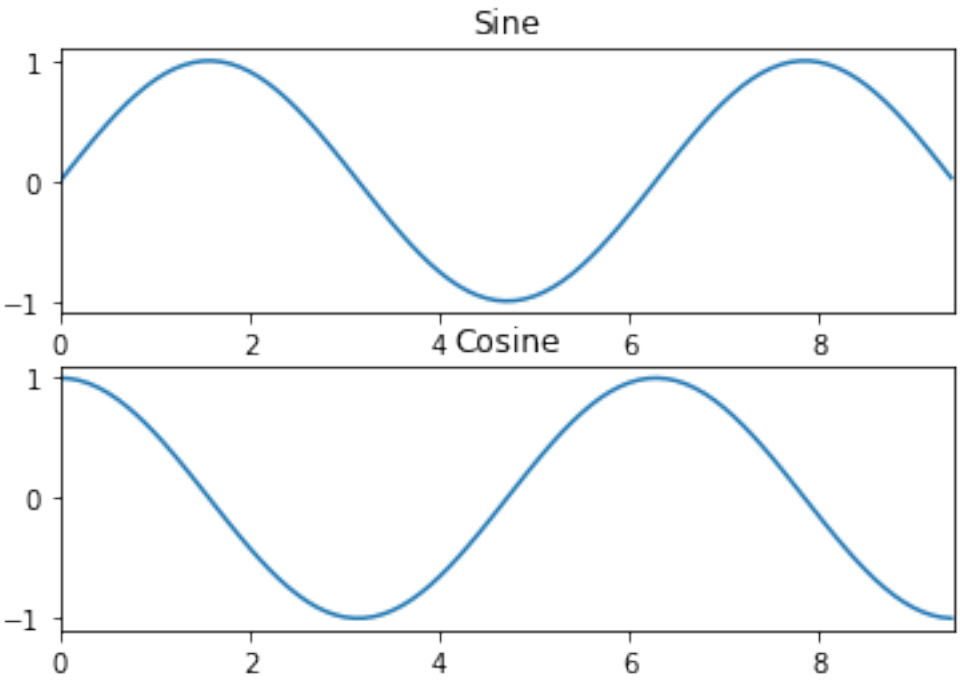
6.7.1 You can display different plots in the same figure using the subplot function. Here is an example:

In [288]:
```
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(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.xlim(0, 3 * np.pi)
plt.title('Sine')

plt.show()
```



6.7.2 26. Complete the above code to create a second subplot in the same grid representing the cosine function. This is how your result should look like.

```
In [289]: 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.xlim(0, 3 * np.pi)
          plt.title('Sine')

          plt.subplot(2, 1, 2)

          plt.plot(x, y_cos)
          plt.xlim(0, 3 * np.pi)
          plt.title('Cosine')



          plt.show()
```



6.8 3.3 Images

6.8.1 27. From your terminal, in your pod session, create an assets directory and transfer the image cat.jpg into it

6.8.2 28. Run the following code and report your result.

```
In [290]: import numpy as np
          import imageio
          import matplotlib.pyplot as plt

          img = imageio.imread('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()
```

The left image is more blue. You can see the sky over the cat's head is more blue in the left image. Furthermore, the right image is more orange. This is because both the green and blue components of the image are reduced in the right side. Therefore, the red component is more noticable in comparison and the right picture seems more orange