Python 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 (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
- IPython: Creating notebooks, Typical workflows

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 [2]: 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([3,6,8,10,1,2,1]))

        [1, 1, 2, 3, 6, 8, 10]
```

Python versions

There are currently two different supported versions of Python, 2.7 and 3.4. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.4 and vice versa. For this class all code will use Python 2.7.

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 [1]: x = 3.0
        print(x, type(x))

        3.0 <class 'float'>
```

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

        4.0
        2.0
        6.0
        27.0
```

```
In [3]: x += 1
        print(x)   # Prints "4"
        x *= 2
        print(x)   # Prints "8"

        4.0
        8.0
```

```
In [4]: y = 2.5
        print(type(y)) # Prints "<type 'float'>"
        print(y, y + 1, y * 2, y ** 3) # Prints "2.5 3.5 5.0 6.25"

        <class 'float'>
        2.5 3.5 5.0 15.625
```

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 (https://docs.python.org/2/library/stdtypes.html#numeric-types-int-float-long-complex).

Booleans

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

```
In [5]: t, f = True, False
        print(type(t)) # Prints "<type 'bool'>"
        print(int(t))

        <class 'bool'>
        1
```

Now we let's look at the operations:

```
In [6]: 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 [7]: hello = 'hello'   # String literals can use single quotes
        world = "world"   # or double quotes; it does not matter.
        print(hello, len(hello))

        hello 5
```

```
In [8]: hw = hello + ' ' + world  # String concatenation
        print(hw)  # prints "hello world"

        hello world
```

```
In [9]: 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 [10]: 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 "
           hello"
         print(s.center(7))    # Center a string, padding with spaces; prints " hello "
         print(s.replace('l', '(ell)'))  # Replace all instances of one substring with
          another;
                                         # prints "he(ell)(ell)o"
         print('  world     '.strip())  # Strip leading and trailing whitespace; prints
           "world"
```

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

You can find a list of all string methods in the documentation
(https://docs.python.org/2/library/stdtypes.html#string-methods).

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

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

```
In [12]: xs[2] = 'foo'     # Lists can contain elements of different types
         print(xs)
```

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

In [13]:
```
xs.append('bar') # Add a new element to the end of the list
print(xs)
```

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

In [14]:
```
x = xs.pop()       # Remove and return the last element of the list
print(x, xs)
```

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

As usual, you can find all the gory details about lists in the underline documentation
(https://docs.python.org/2/tutorial/datastructures.html#more-on-lists).

Slicing

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

In [15]:
```
nums = range(5)     # range is a built-in function that creates a list of integ
ers
print(nums)          # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])      # Get a slice from index 2 to 4 (exclusive); prints "[2,
 3]"
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
ts "[0, 1]"
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]"
```

```
range(0, 5)
range(2, 4)
range(2, 5)
range(0, 2)
range(0, 5)
range(0, 4)
range(0, 5)
```

Loops

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

```
In [16]:  animals = ['cat', 'dog', 'monkey']
          for animal1 in animals:
              print(animal1)

          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 [17]:  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 [18]:  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 [19]:  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 [20]:  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 [21]: d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
         print(d['cat'])          # Get an entry from a dictionary; prints "cute"
         print('cat' in d)        # Check if a dictionary has a given key; prints "True"

         cute
         True
```

```
In [22]: d['fish'] = 'wet'      # Set an entry in a dictionary
         print(d['fish'])        # Prints "wet"

         wet
```

```
In [23]: print(d['monkey'])     # KeyError: 'monkey' not a key of d

         ---------------------------------------------------------------------------
         KeyError                                  Traceback (most recent call last)
         <ipython-input-23-78265c35c053> in <module>()
         ----> 1 print(d['monkey'])   # KeyError: 'monkey' not a key of d

         KeyError: 'monkey'
```

```
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"
```

```
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"
```

You can find all you need to know about dictionaries in the documentation (https://docs.python.org/2/library/stdtypes.html#dict).

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

```
In [24]: 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 [25]:  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 [26]:  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}
```

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 [27]:  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[t])
          print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

```
In [28]:  import numpy as np
          np.arange(30).reshape((t))
```

```
Out[28]:  array([[ 0,  1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10, 11],
                 [12, 13, 14, 15, 16, 17],
                 [18, 19, 20, 21, 22, 23],
                 [24, 25, 26, 27, 28, 29]])
```

Functions

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

```
In [29]:  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 [30]:  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 [31]:  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 (http://wiki.scipy.org/NumPy_for_Matlab_Users) useful to get started with Numpy.

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

```
In [32]:  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 [33]: 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 [34]: b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
         print(b)
```

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

```
In [35]: 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 [36]: a = np.zeros(1)  # Create an array of all zeros
         print(a)
```

```
[ 0.]
```

```
In [37]: b = np.ones((1,2))    # Create an array of all ones
         print(b)
```

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

```
In [38]: c = np.full((2,2), 7) # Create a constant array
         print(c)
```

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

```
In [39]: d = np.eye(2)        # Create a 2x2 identity matrix
         print(d)
```

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

```
In [40]: e = np.random.rand(2,2)# Create an array filled with random values
         print(e)
```

```
[[ 0.1525549   0.09101277]
 [ 0.56593666  0.20834519]]
```

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 [41]:  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):
          # [[2 3]
          #  [6 7]]
          b = a[:2, 1:3]
          print(b)

          [[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 [42]:  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 [43]:  # 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 [44]:  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 [45]:  # 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()
          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 [46]:  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 [47]:  # 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 [48]:  # 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 [49]:  # 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 [50]:  # 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]]
```

```
In [51]:  np.arange(4).reshape((4,1))
```

```
Out[51]:  array([[0],
                 [1],
                 [2],
                 [3]])
```

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 [52]:  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 [53]:  # 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]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

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 [54]:  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.float32)  # Force a particular datatype

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

```
int32 float64 float32
```

You can read all about numpy datatypes in the documentation (http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html).

Array math

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

```
In [55]:  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(np.add(x, y))
```

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

```
In [56]:  ok = [4,5,2]
          ss = [2,1,2]
          bb = ss + ok
          print(bb)
```

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

```
In [57]:  ok = np.array([4,5,2])
          ss = np.array([2,1,2])
          print(np.add(ok,ss))
```

```
[6 6 4]
```

```
In [58]:  # Elementwise difference; both produce the array
          print(x - y)
          print(np.subtract(x, y))
```

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

```
In [59]:  # Elementwise product; both produce the array
          print(x * y)
          print(np.multiply(x, y))
```

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

```
In [60]:  # 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 [61]:  # 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 [62]:  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(np.dot(x, y))
```

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

```
In [63]:  # 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 [64]:  # 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 [65]:  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 (http://docs.scipy.org/doc/numpy/reference/routines.math.html).

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 [68]:  print(x)
          print(x.T)

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

```
In [69]:  v = np.array([[1,2,3]])
          print(v)
          print(v.T)

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

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 (http://docs.scipy.org/doc/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 [70]:  import matplotlib.pyplot as plt
```

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

```
In [71]:  %matplotlib inline
```

Plotting

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

In [72]:
```python
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
print(x)
y = np.sin(x)
# Plot the points using matplotlib
plt.plot(y)
```

```
[ 0.    0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3  1.4
  1.5  1.6  1.7  1.8  1.9  2.   2.1  2.2  2.3  2.4  2.5  2.6  2.7  2.8  2.9
  3.   3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4.   4.1  4.2  4.3  4.4
  4.5  4.6  4.7  4.8  4.9  5.   5.1  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9
  6.   6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9  7.   7.1  7.2  7.3  7.4
  7.5  7.6  7.7  7.8  7.9  8.   8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9
  9.   9.1  9.2  9.3  9.4]
```
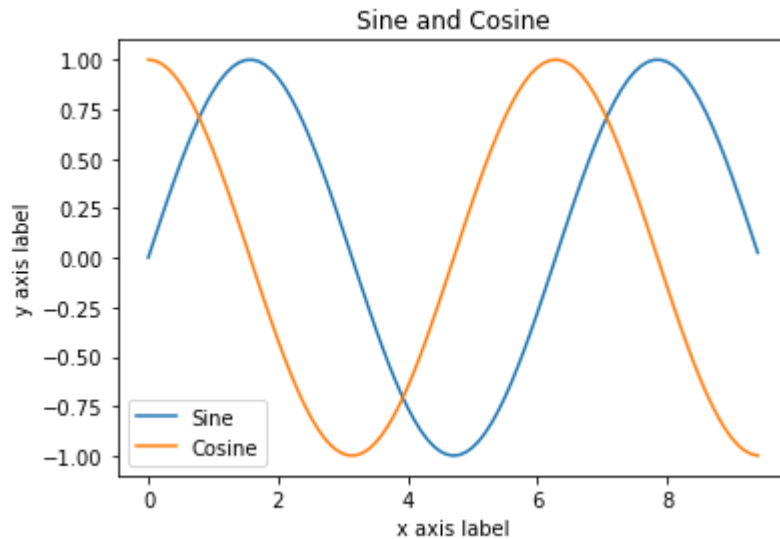
Out[72]: [<matplotlib.lines.Line2D at 0x99ba0f0>]
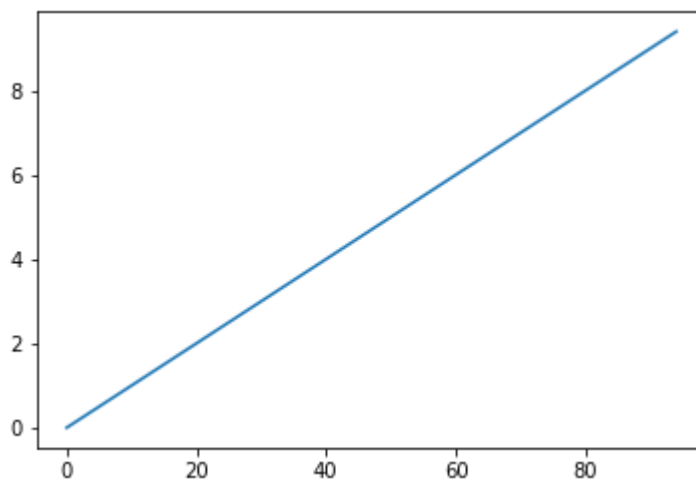
With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

```
In [73]: 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'])
```

Out[73]: <matplotlib.legend.Legend at 0x9a48390>

Subplots

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

```
In [74]:  # 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()
```
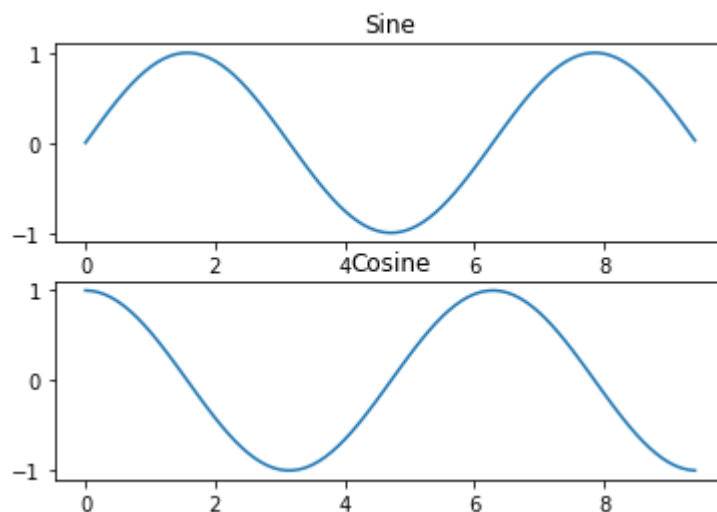
You can read much more about the `subplot` function in the documentation
(http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.subplot).