# Scientific Python 101

Shan-Hung Wu & DataLab
Fall 2025

Python is one of the most popular programming languages for data science.

This lab guides you through basics of Python for the Deep Learning course and provides some useful references.

NOTE: all code examples provided in this course are written in Python 3, which is not compatible with Python 2. Please make sure that you know about the major differences between the two Python versions.

Python enjoys a large number of useful add-on packages developed by its active community. For this tutorial, we will use the following packages:

- NumPy: provides multi-dimensional arrays to store and manipulate data;
- SciPy: built on top of NumPy that provides a large number of functions operating on `Numpy` arrays and useful for scientific applications;
- Pandas: built on top of NumPy that provides additional higher level data manipulation tools that make working with tabular data even more convenient;
- Scikit-learn: provides the off-the-shelf implementation of various machine learning algorithms;
- Matplotlib and Seaborn: provides various plotting functions for data and model visualization;
- Tensorflow: for deep learning.

# Environment Setup

The easiest way to setting up a scientific Python environment is to install the Anaconda for Python 3. Anaconda is a free Python distribution that bundles all the essential Python packages for data science, math, and engineering in one user-friendly cross-platform distribution.

## Package Management

Anaconda comes with Conda, a package manager application that quickly installs, runs, and updates packages and their dependencies. To list all installed packages, type the following in the command-line:

```
> conda list
```

To install a new package, type:

```
> conda install package-name
```

To update an existing package, type:

```
> conda update package-name
```

You can also update Python with the `update` command:

```
> conda update python
```

## Environment Management

Sometimes, you need to switch to an environment with a specific Python version and its packages. You don't have to re-install Python if your version does not match. Instead, you can simply use Conda to manage multiple environments. To create a new Python environment, say version 3.6, simply type:

```
> conda create -n env-name python=3.6
```

You can switch to an environment by typing:

```
> source activate env-name (Linux or Mac)
> activate env-name (Windows)
```

The Python version you choose when downloading `Anaconda` is called the `root` environment. To list all environments, type:

```
> conda info --envs
```

NOTE: each environment maintains the list of installed packages independently. So, you may find that some packages are missing in a newly created environment. If this happens, simply install the missing package using the `conda install` command in that environment.

## Virtualenv

An alternative to using Anaconda is to install virtualenv using pip. Before running the following commands, make sure you have Python and pip installed.

To install virtualenv, type the following in the command-line:

```
> pip install virtualenv
```

After the installation, you can create a Python 3 virtual environment:

```
> virtualenv -p python3 env-name
```

To activate the created environment:

```
> source activate env-name (Linux or Mac)
> activate env-name (Windows)
```

Install Jupyter packages:

```
> pip install ipython
> pip install jupyter
```

You can open jupyter notebook by running:

```
> jupyter notebook
```

And also the packages we are going to use:

```
> pip install numpy
> pip install scipy
> pip install Pillow
> pip install matplotlib
```

# Opening Lab Notes and Running Code

All lab notes, including this doc, are written and released in the Jupyter Notebook format (with file extension `*.ipynb`). Jupyter Notebook is a web application that allows you to create and share documents that contain live Python code, $\LaTeX$ equations, visualizations, and rich explanatory text. We highly recommend you to play with the code in a notebook. To open a notebook, switch to the directory containing the notebook in the command-line and start the Jupyter web server (locating at `http://127.0.0.1:8888` by default) by typing:

```
> jupyter notebook
```

Then, a web page will be opened automatically. By clicking the notebook name, you open the notebook in a web page. Now, you can re-run a code snippet by hitting `Shift+Enter` in a code cell. You can also edit some text using the lightweight Markdown language in a Markdown cell.

Sometimes, you wish to measure the execution time of a code cell. You can do this by adding the following as the first line of the code:

```
%%timeit
```

This is one of the built-in magic commands.

# Readings

If you are not familiar with Python yet, you may read this tutorial.

## Scientific Python

To have an overview of the scientific packages like NumPy, SciPy, and Matlibplot, we will follow this tutorial (CS231n: Convolutional Neural Networks for Visual Recognition, Stanford) for this class. If you want to see the differences between Python 2 and 3, check this source.

## Calculus

Don't forget to grab your favorite Calculus book and read again the topics about **multivariate** calculus:

- Product rules, quotient rules, and chain rules;
- Partial derivatives and directional derivatives;
- Gradient of a (multivariate) function;
- Jacobian matrix of a vector-valued function;
- Hessian matrix
- Taylor's theorem.

If you don't know where to start, you can follow Stanford CS 229 - Linear Algebra and Calculus refresher and Jacobian matrix and determinant.

If you think you have a good understanding of the above topics, congrats! You are now ready for the **Deep Learning** course!

# Basic Python

In [1]:
```
print ("Hello world!")
```

```
Hello world!
```

## Basic data types

### Numbers

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

In [2]:
```
x = 3
print (x), type(x)
```

```
3
```

Out[2]:
```
(None, int)
```

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

```
4
2
6
9
```

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

```
4
8
```

In [5]:
```
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 [6]:
```
t, f = True, False
print (type(t)) # Prints "<type 'bool'>"
```

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

Now we let's look at the operations:

In [7]:
```
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 [8]:
```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print (hello, len(hello))
```

```
hello 5
```

In [9]:
```
hw = hello + ' ' + world  # String concatenation
print (hw)  # prints "hello world"
```

```
hello world
```

In [10]:
```
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 [11]:
```
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.

## 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 [12]:
```
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 [13]:
```
xs[2] = 'foo'    # Lists can contain elements of different types
print (xs)
```
```
[3, 1, 'foo']
```

In [14]:
```
xs.append('bar') # Add a new element to the end of the list
print (xs)
```
```
[3, 1, 'foo', 'bar']
```

In [15]:
```
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 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 [16]:
```
nums = list(range(5))     # range is a built-in function that creates a list of integers
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); prints "[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)
```
```
[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 [17]:
```
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 [18]:
```
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 [19]:
```
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 [20]:
```
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 [21]:
```
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 [22]:
```
# Create a new dictionary with some data
d = {'cat': 'cute', 'dog': 'furry'}
# Get an entry from a dictionary; prints "cute"
print (d['cat'])
# Check if a dictionary has a given key; prints "True"
print ('cat' in d)
```

```
cute
True
```

In [23]:
```
d['fish'] = 'wet'    # Set an entry in a dictionary
print (d['fish'])      # Prints "wet"
```

```
wet
```

In [24]:
```
#D_24 = d
#d = D_24
print (d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-24-205072c442cc> in <module>()
      1 #D_24 = d
      2 #d = D_24
----> 3 print (d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

In [25]:
```
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"
```

```
N/A
wet
```

In [26]:
```
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 [27]:
```
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 items method:

In [28]:
```
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 [29]:
```
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 [30]:
```
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 [31]:
```
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 [32]:
```
animals.add('cat')       # Adding an element that is already in the set does nothing
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 [33]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print ('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

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

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

In [34]:
```
from math import sqrt
print ({int(sqrt(x)) for x in range(30)})
```

```
{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 [35]:
```
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
```

The documentation has more information about tuples.

## Functions

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

In [36]:
```
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 [37]:
```
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 [38]:
```
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 [39]:
```
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 [40]:
```
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 [41]:
```
b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print (b)
```

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

In [42]:
```
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 [43]:
```
a = np.zeros((2,2))  # Create an array of all zeros
print (a)
```

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

In [44]:
```
b = np.ones((1,2))   # Create an array of all ones
print (b)
```

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

In [45]:
```
c = np.full((2,2), 7) # Create a constant array
print (c)
```

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

In [46]:
```
d = np.eye(2)        # Create a 2x2 identity matrix
print (d)
```

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

In [47]:
```
e = np.random.random((2,2)) # Create an array filled with random values
print (e)
```

```
[[0.72546598 0.60528063]
 [0.54633336 0.94981052]]
```

## 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 [48]:
```
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 [49]:
```
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 [50]:
```
# 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 [51]:
```
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 [52]:
```
# 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 [53]:
```
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 [54]:
```
# 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 [55]:
```
# 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 [56]:
```
# 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 [57]:
```
# 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 [58]:
```
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 [59]:
```
# 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 [60]:
```
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 [61]:
```
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
# [[ 6.0  8.0]
#  [10.0 12.0]]
print (x + y)
print (np.add(x, y))
```

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

In [62]:
```
# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print (x - y)
print (np.subtract(x, y))
```

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

In [63]:
```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print (x * y)
print (np.multiply(x, y))
```

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

In [64]:
```
# 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 [65]:
```
# 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 [66]:
```
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 [67]:
```
# 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 [68]:
```
# 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 [69]:
```
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 [70]:
```
print (x)
print (x.T)
```

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

In [71]:
```
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 [72]:
```
# 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 [73]:
```
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 [74]:
```
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 [75]:
```
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 [76]:
```
# 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 [77]:
```
# 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 [78]:
```
# 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 [79]:
```
# 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 [80]:
```
# 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 [81]:
```
# Watch out for this. Adding a column vector to row vector returns a matrix
# We would expect to get an error like "dimensions not matching", but broadcating allows this addition.

test = np.array([[1,2,3]])
test2 = np.array([[4],[5],[6]])

print(test)
print(test2)
print(test+test2)
```

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

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.

# Scipy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

The best way to get familiar with SciPy is to browse the documentation. We will highlight some parts of SciPy that you might find useful for this class.

## Image operations

SciPy provides some basic functions to work with images. For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images. Here is a simple example that showcases these functions:

In [82]:
```
from PIL import Image

# Read an JPEG image into a numpy array
# Note: Assuming you have a folder assets with an image to work with
img = np.asarray(Image.open('assets/cat.jpg'))
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = (img.astype(np.float32) * [1, 0.95, 0.9])

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = Image.fromarray(img_tinted.astype(np.uint8), "RGB").resize((300, 300))

# Write the tinted image back to disk
img_tinted.save('assets/cat_tinted.jpg')
```

```
uint8 (400, 248, 3)
```

Top: The original image. Bottom: The tinted and resized image.

## Distance between points

SciPy defines some useful functions for computing distances between sets of points.

The function `scipy.spatial.distance.pdist` computes the distance between all pairs of points in a given set:

In [83]:
```
import numpy as np
from scipy.spatial.distance import pdist, squareform

# Create the following array where each row is a point in 2D space:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

```
[[0 1]
 [1 0]
 [2 0]]
[[0.         1.41421356 2.23606798]
 [1.41421356 0.         1.        ]
 [2.23606798 1.         0.        ]]
```

You can read all the details about this function in the documentation.

A similar function ( `scipy.spatial.distance.cdist` ) computes the distance between all pairs across two sets of points; you can read about it in the documentation.

# 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.

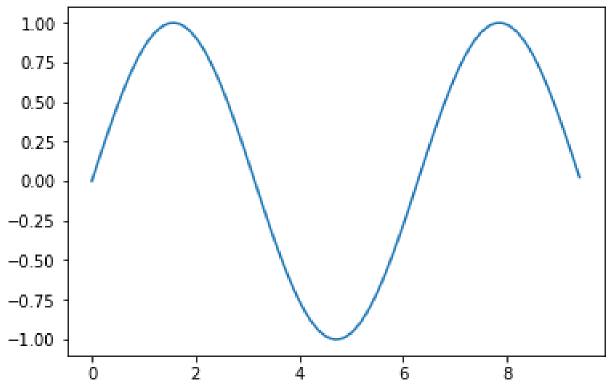
By running this special iPython command, we will be displaying plots inline:

In [84]:
```
%matplotlib inline
```

## Plotting

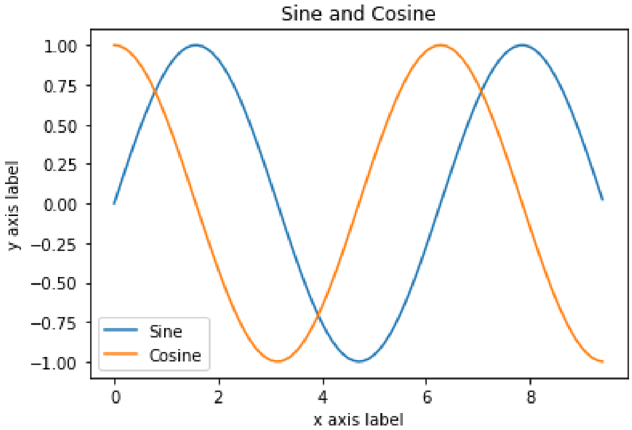
The most important function in `matplotlib` is plot, which allows you to plot 2D data. Here is a simple example:

In [85]:
```
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.
```




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

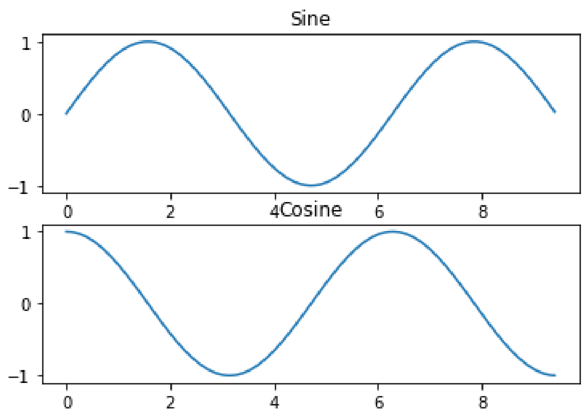
In [86]:
```
import numpy as np
import matplotlib.pyplot as plt

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

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




## Subplots

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

In [87]:
```
import numpy as np
import matplotlib.pyplot as plt

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

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

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

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

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




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