▼ 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
- IPython: Creating notebooks, Typical workflows

▼ A Brief Note on Python Versions

As of Janurary 1, 2020, Python has officially dropped support for `python2`. We'll be using Python 3.7 for this iteration of the course. You can check your Python version at the command line by running `python --version`. In Colab, we can enforce the Python version by clicking `Runtime -> Change Runtime Type` and selecting `python3`. Note that as of April 2020, Colab uses Python 3.6.9 which should run everything without any errors.

```
1 !python --version
```

```
Python 3.6.9
```

▼ 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:

```
 1 def quicksort(arr):
 2     if len(arr) <= 1:
 3         return arr
 4     pivot = arr[len(arr) // 2]
 5     left = [x for x in arr if x < pivot]
 6     middle = [x for x in arr if x == pivot]
 7     right = [x for x in arr if x > pivot]
 8     return quicksort(left) + middle + quicksort(right)
 9
10 print(quicksort([3,6,8,10,1,2,1]))
```

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

Basic data types

Numbers

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

```
1 x = 3
2 print(x, type(x))
```

    3 <class 'int'>

```
1 print(x + 1)   # Addition
2 print(x - 1)   # Subtraction
3 print(x * 2)   # Multiplication
4 print(x ** 2)  # Exponentiation
```

    4
    2
    6
    9

```
1 x += 1
2 print(x)
3 x *= 2
4 print(x)
```

    4
    8

```
1 y = 2.5
2 print(type(y))
3 print(y, y + 1, y * 2, y ** 2)
```

    <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.):

```
1 t, f = True, False
2 print(type(t))
```

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

Now we let's look at the operations:

```
1 print(t and f) # Logical AND;
2 print(t or f)  # Logical OR;
3 print(not t)   # Logical NOT;
4 print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings

```
1 hello = 'hello'   # String literals can use single quotes
2 world = "world"   # or double quotes; it does not matter
3 print(hello, len(hello))
```

```
hello 5
```

```
1 hw = hello + ' ' + world  # String concatenation
2 print(hw)
```

```
hello world
```

```
1 hw12 = '{} {} {}'.format(hello, world, 12)  # string formatting
2 print(hw12)
```

```
hello world 12
```

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

```
1 s = "hello"
2 print(s.capitalize())  # Capitalize a string
3 print(s.upper())       # Convert a string to uppercase; prints "HELLO"
4 print(s.rjust(7))      # Right-justify a string, padding with spaces
5 print(s.center(7))     # Center a string, padding with spaces
6 print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another
7 print('  world '.strip())  # Strip leading and trailing whitespace
```

```
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:

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

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

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

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

```
1 xs.append('bar') # Add a new element to the end of the list
2 print(xs)
```

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

```
1 x = xs.pop()     # Remove and return the last element of the list
```

```
1 x   = xs.pop()       # Remove and return the last element of the list
2 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:

```
1 nums = list(range(5))     # range is a built-in function that creates a list of integers
2 print(nums)          # Prints "[0, 1, 2, 3, 4]"
3 print(nums[2:4])     # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
4 print(nums[2:])      # Get a slice from index 2 to the end; prints "[2, 3, 4]"
5 print(nums[:2])      # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
6 print(nums[:])       # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
7 print(nums[:-1])     # Slice indices can be negative; prints ["0, 1, 2, 3]"
8 nums[2:4] = [8, 9] # Assign a new sublist to a slice
9 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:

```
1 animals = ['cat', 'dog', 'monkey']
2 for animal in animals:
3     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:

```
1 animals = ['cat', 'dog', 'monkey']
2 for idx, animal in enumerate(animals):
3     print('#{}: {}'.format(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:

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

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

You can make this code simpler using a list comprehension:

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

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

List comprehensions can also contain conditions:

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

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

```
cute
True
```

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

wet

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

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-26-78fc9745d9cf> in <module>()
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

    SEARCH STACK OVERFLOW

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

N/A
wet

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

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-29-6d73d138ca67> in <module>()
----> 1 del d['fish']          # Remove an element from a dictionary
      2 print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"

KeyError: 'fish'
```

    SEARCH STACK OVERFLOW

You can find all you need to know about dictionaries in the [documentation](#).

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

```
1 d = {'person': 2, 'cat': 4, 'spider': 8}
2 for animal, legs in d.items():
3     print('A {} has {} legs'.format(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:

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

```
1 animals = {'cat', 'dog'}
2 print('cat' in animals)   # Check if an element is in a set; prints "True"
3 print('fish' in animals)  # prints "False"
4
```

```
True
False
```

```
1 animals.add('fish')      # Add an element to a set
2 print('fish' in animals)
3 print(len(animals))       # Number of elements in a set;
```

```
True
3
```

```
1 animals.add('cat')       # Adding an element that is already in the set does nothing
2 print(len(animals))
3 animals.remove('cat')    # Remove an element from a set
4 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:

```
1 animals = {'cat', 'dog', 'fish'}
2 for idx, animal in enumerate(animals):
3     print('#{}: {}'.format(idx + 1, animal))
```

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

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

```
1 from math import sqrt
2 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:

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

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

```
1 t[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-c8aeb8cd20ae> in <module>()
----> 1 t[0] = 1

TypeError: 'tuple' object does not support item assignment
```

SEARCH STACK OVERFLOW

Functions

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

```
1 def sign(x):
2     if x > 0:
3         return 'positive'
4     elif x < 0:
5         return 'negative'
6     else:
7         return 'zero'
8
9 for x in [-1, 0, 1]:
10    print(sign(x))
```

    negative
    zero
    positive

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

```
1 def hello(name, loud=False):
2     if loud:
3         print('HELLO, {}'.format(name.upper()))
4     else:
5         print('Hello, {}!'.format(name))
6
7 hello('Bob')
8 hello('Fred', loud=True)
```

    Hello, Bob!
    HELLO, FRED

Classes

The syntax for defining classes in Python is straightforward:

```
1 class Greeter:
2
3     # Constructor
4     def __init__(self, name):
5         self.name = name  # Create an instance variable
6
7     # Instance method
8     def greet(self, loud=False):
9         if loud:
10          print('HELLO, {}'.format(self.name.upper()))
11        else:
12          print('Hello, {}!'.format(self.name))
13
14 g = Greeter('Fred')  # Construct an instance of the Greeter class
15 g.greet()            # Call an instance method; prints "Hello, Fred"
16 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:

```
1 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:

```
1 a = np.array([1, 2, 3])   # Create a rank 1 array
2 print(type(a), a.shape, a[0], a[1], a[2])
3 a[0] = 5                   # Change an element of the array
4 print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

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

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

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

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

Numpy also provides many functions to create arrays:

```
1 a = np.zeros((2,2))  # Create an array of all zeros
2 print(a)
```

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

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

> [[1. 1.]]

```
1 c = np.full((2,2), 7) # Create a constant array
2 print(c)
```

> [[7 7]
> [7 7]]

```
1 d = np.eye(2)          # Create a 2x2 identity matrix
2 print(d)
```

> [[1. 0.]
> [0. 1.]]

```
1 e = np.random.random((2,2)) # Create an array filled with random values
2 print(e)
```

> [[0.91050427 0.40429241]
> [0.4859996 0.86128849]]

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:

```
1 import numpy as np
2
3 # Create the following rank 2 array with shape (3, 4)
4 # [[ 1  2  3  4]
5 #  [ 5  6  7  8]
6 #  [ 9 10 11 12]]
7 a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
8
9 # Use slicing to pull out the subarray consisting of the first 2 rows
10 # and columns 1 and 2; b is the following array of shape (2, 2):
11 # [[2 3]
12 #  [6 7]]
13 b = a[:2, 1:3]
14 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.

```
1 print(a[0, 1])
```

```
2 b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
3 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:

```
1 # Create the following rank 2 array with shape (3, 4)
2 a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
3 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:

```
1 row_r1 = a[1, :]    # Rank 1 view of the second row of a
2 row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
3 row_r3 = a[[1], :]  # Rank 2 view of the second row of a
4 print(row_r1, row_r1.shape)
5 print(row_r2, row_r2.shape)
6 print(row_r3, row_r3.shape)
```

> [5 6 7 8] (4,)
> [[5 6 7 8]] (1, 4)
> [[5 6 7 8]] (1, 4)

```
1 # We can make the same distinction when accessing columns of an array:
2 col_r1 = a[:, 1]
3 col_r2 = a[:, 1:2]
4 print(col_r1, col_r1.shape)
5 print()
6 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:

```
1 a = np.array([[1,2], [3, 4], [5, 6]])
2
3 # An example of integer array indexing.
4 # The returned array will have shape (3,) and
5 print(a[[0, 1, 2], [0, 1, 0]])
6
7 # The above example of integer array indexing is equivalent to this:
8 print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

    [1 4 5]
    [1 4 5]

```
1 # When using integer array indexing, you can reuse the same
2 # element from the source array:
3 print(a[[0, 0], [1, 1]])
4
5 # Equivalent to the previous integer array indexing example
6 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:

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

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

```
1 # Create an array of indices
2 b = np.array([0, 2, 0, 1])
3
4 # Select one element from each row of a using the indices in b
5 print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

    [1 6 7 11]

```
1 # Mutate one element from each row of a using the indices in b
2 a[np.arange(4), b] += 10
3 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:

```
 1 import numpy as np
 2
 3 a = np.array([[1,2], [3, 4], [5, 6]])
 4
 5 bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
 6                      # this returns a numpy array of Booleans of the same
 7                      # shape as a, where each slot of bool_idx tells
 8                      # whether that element of a is > 2.
 9
10 print(bool_idx)
```

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

```
 1 # We use boolean array indexing to construct a rank 1 array
 2 # consisting of the elements of a corresponding to the True values
 3 # of bool_idx
 4 print(a[bool_idx])
 5
 6 # We can do all of the above in a single concise statement:
 7 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:

```
 1 x = np.array([1, 2])  # Let numpy choose the datatype
 2 y = np.array([1.0, 2.0])  # Let numpy choose the datatype
 3 z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
 4
 5 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:

```
1 x = np.array([[1,2],[3,4]], dtype=np.float64)
2 y = np.array([[5,6],[7,8]], dtype=np.float64)
3
4 # Elementwise sum; both produce the array
5 print(x + y)
6 print(np.add(x, y))
```

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

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

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

```
1 # Elementwise product; both produce the array
2 print(x * y)
3 print(np.multiply(x, y))
```

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

```
1 # Elementwise division; both produce the array
2 # [[ 0.2        0.33333333]
3 #  [ 0.42857143 0.5       ]]
4 print(x / y)
5 print(np.divide(x, y))
```

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

```
1 # Elementwise square root; produces the array
```

```
1 # Elementwise square root; produces the array
2 # [[ 1.          1.41421356]
3 #  [ 1.73205081  2.         ]]
4 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:

```
1 x = np.array([[1,2],[3,4]])
2 y = np.array([[5,6],[7,8]])
3
4 v = np.array([9,10])
5 w = np.array([11, 12])
6
7 # Inner product of vectors; both produce 219
8 print(v.dot(w))
9 print(np.dot(v, w))
```

```
219
219
```

You can also use the `@` operator which is equivalent to numpy's `dot` operator.

```
1 print(v @ w)
```

```
219
```

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

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

```
1 # Matrix / matrix product; both produce the rank 2 array
2 # [[19 22]
3 #  [43 50]]
4 print(x.dot(y))
5 print(np.dot(x, y))
6 print(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`:

```
1 x = np.array([[1,2],[3,4]])
2
3 print(np.sum(x))  # Compute sum of all elements; prints "10"
4 print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
5 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:

```
1 print(x)
2 print("transpose\n", x.T)
```

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

```
1 v = np.array([[1,2,3]])
2 print(v )
3 print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[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:

```
1 # We will add the vector v to each row of the matrix x,
2 # storing the result in the matrix y
3 x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
4 v = np.array([1, 0, 1])
5 y = np.empty_like(x)   # Create an empty matrix with the same shape as x
6
7 # Add the vector v to each row of the matrix x with an explicit loop
8 for i in range(4):
9     y[i, :] = x[i, :] + v
10
11 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:

```
1 vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
2 print(vv)                # Prints "[[1 0 1]
3                          #          [1 0 1]
4                          #          [1 0 1]
5                          #          [1 0 1]]"
```

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

```
1 y = x + vv  # Add x and vv elementwise
2 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:

```
1 import numpy as np
2
```

```
3 # We will add the vector v to each row of the matrix x,
4 # storing the result in the matrix y
5 x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
6 v = np.array([1, 0, 1])
7 y = x + v  # Add v to each row of x using broadcasting
8 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:

```
1 # Compute outer product of vectors
2 v = np.array([1,2,3])  # v has shape (3,)
3 w = np.array([4,5])    # w has shape (2,)
4 # To compute an outer product, we first reshape v to be a column
5 # vector of shape (3, 1); we can then broadcast it against w to yield
6 # an output of shape (3, 2), which is the outer product of v and w:
7
8 print(np.reshape(v, (3, 1)) * w)
```

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

```
1 # Add a vector to each row of a matrix
```

```
2 x = np.array([[1,2,3], [4,5,6]])
3 # x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
4 # giving the following matrix:
5
6 print(x + v)
```

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

```
1 # Add a vector to each column of a matrix
2 # x has shape (2, 3) and w has shape (2,).
3 # If we transpose x then it has shape (3, 2) and can be broadcast
4 # against w to yield a result of shape (3, 2); transposing this result
5 # yields the final result of shape (2, 3) which is the matrix x with
6 # the vector w added to each column. Gives the following matrix:
7
8 print((x.T + w).T)
```

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

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

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

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

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

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`

```
1 import matplotlib.pyplot as plt
```

By running this special iPython command, we will be displaying plots inline:
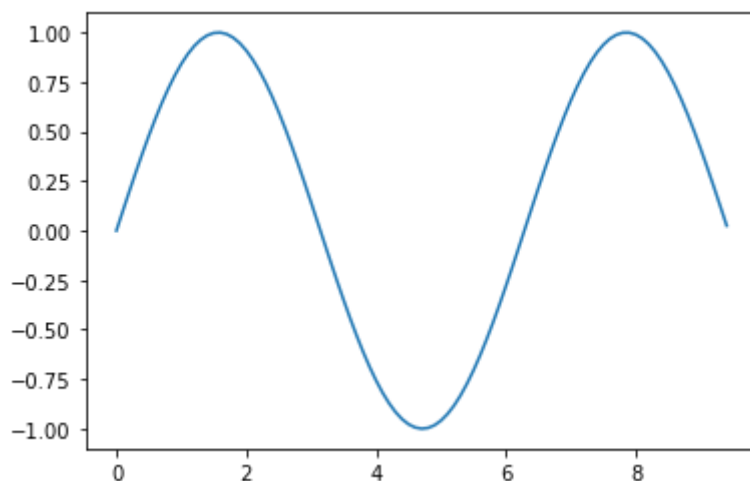
```
1 %matplotlib inline
```

▾ Plotting

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

```
1 # Compute the x and y coordinates for points on a sine curve
2 x = np.arange(0, 3 * np.pi, 0.1)
3 y = np.sin(x)
4
5 # Plot the points using matplotlib
6 plt.plot(x, y)
```
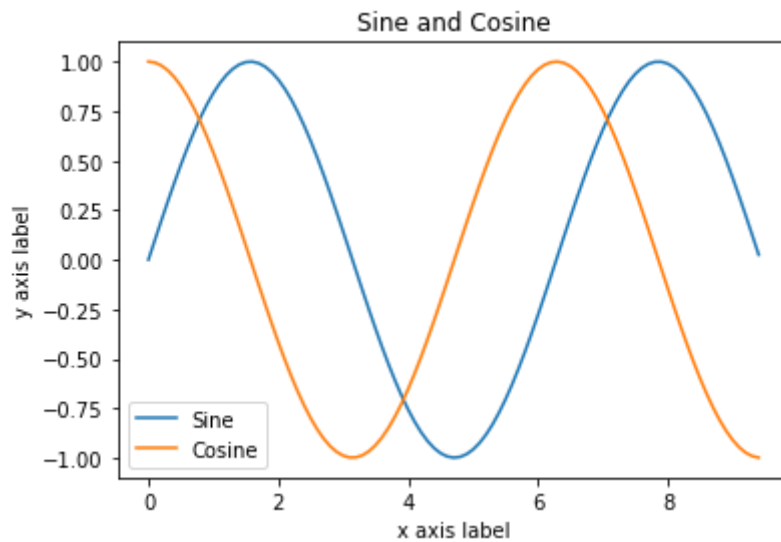
[<matplotlib.lines.Line2D at 0x7fa2f674c0b8>]

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

```
1 y_sin = np.sin(x)
2 y_cos = np.cos(x)
3
4 # Plot the points using matplotlib
5 plt.plot(x, y_sin)
6 plt.plot(x, y_cos)
7 plt.xlabel('x axis label')
8 plt.ylabel('y axis label')
9 plt.title('Sine and Cosine')
10 plt.legend(['Sine', 'Cosine'])
```

<matplotlib.legend.Legend at 0x7fa2f66cc780>
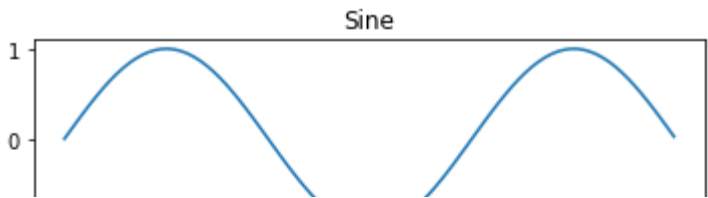

Subplots

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

```
1  # Compute the x and y coordinates for points on sine and cosine curves
2  x = np.arange(0, 3 * np.pi, 0.1)
3  y_sin = np.sin(x)
4  y_cos = np.cos(x)
5
6  # Set up a subplot grid that has height 2 and width 1,
7  # and set the first such subplot as active.
8  plt.subplot(2, 1, 1)
9
10 # Make the first plot
11 plt.plot(x, y_sin)
12 plt.title('Sine')
13
14 # Set the second subplot as active, and make the second plot.
15 plt.subplot(2, 1, 2)
16 plt.plot(x, y_cos)
17 plt.title('Cosine')
18
19 # Show the figure.
20 plt.show()
```

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