Python3 Tutorial

Aug 17, 2019

Adapted by Badri Adhikari from the original adaptation by Volodymyr Kuleshov and Isaac Caswell from the `CS231n` Python tutorial by Justin Johnson (http://cs231n.github.io/python-numpy-tutorial/). Expanded the broadcasting section by adapting this 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.

This tutorial will serve as a quick crash course **both** on the Python programming language and on the use of Python for scientific computing.

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

Basics of Python

Python versions

- Two versions of Python are widely used - 2.x and 3.x
- 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
- You can check your Python version at the command line by running `python --version`
- Support for Python2 expires in 2020 - https://pythonclock.org/

```
import platform
print(platform.python_version())
```

```
3.6.8
```

```
# Throws an error in Python3
print'Hello Python'
```

```
  File "<ipython-input-2-3ad3a54c48d4>", line 1
    print'Hello Python'
                      ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

```
# Runs in both Python2 and Python3
print ('Hello Python')
```

```
Hello Python
```

Basic data types

Numbers

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

```
x = 3
print( x )
print( type(x) )
```

```
3
<class 'int'>
```

"`type`" is your screwdriver!

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

```
4
2
6
9
```

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

```
4
8
```

```
y = 2.5
print( type(y) )
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:

- Python 2 has int, **long**, float, and complex data types documentation.
- Python 3 has int, float, and complex data types documentation.

```
# Automatically creates a variable of the required type (in Python2)
x = 100000000000000000
print(type(x))
x = 10000000000000000000
print(type(x))
x = 10.0
print(type(x))
x = -1.0 + 1.0j
print(type(x))
x = '10.0'
print(type(x))
```

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'str'>
```

Python 2 has two integer types int and long. These have been unified in **Python 3**, so there is now **only one type, int**. Read more here.

```
x = long(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-b8f22273a3e3> in <module>()
----> 1 x = long(x)

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

SEARCH STACK OVERFLOW

Booleans

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

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

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

Now we let's look at the operations:

```
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
```

Bitwise vs Logical Operators

```python
print( t & f )  # Bitwise AND (Not usually used)
print( t | f )  # Bitwise OR  (Not usually used)
```

```
False
True
```

```python
print( 0 < 1 & 0 < 2 )
print( 0 < 1 and 0 < 2 )
# This is because the precedence of & and 'and'
# & has higher precedence than <; 'and' has lower precedence than <
```

```
False
True
```

Strings

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

```
hello 5
```

```python
hw = hello + ' ' + world  # String concatenation
print( hw, hello )  # prints "hello world"
```

```
hello world hello
```

```python
print('%13s %13s %d' % (hello, world, 12) )  # sprintf style string formatting
```

```
        hello         world 12
```

```python
print(hello + world + str(12) )
```

```
helloworld12
```

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

```python
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;

print( '  world '.strip() )      # Strip leading and trailing whitespace; prints "world"

s = "hello class   good     evening"
cols = s.split()                 # Automatically detects multiple spaces
print( cols[0], cols[2] )
```

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

String methods 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:

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

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

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

```
[3, 1, 'foo', 2]
```

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

```
[3, 1, 'foo', 2, 'bar']
```

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

```
bar [3, 1, 'foo', 2]
```

```
xs = [3, 1, 5, 1, 3, 2]
xs2 = [10, 11]
xs.append(xs2)
print( xs )
```

```
[3, 1, 5, 1, 3, 2, [10, 11]]
```

```
xs_another = [100, 1000, 200]
xs.extend(xs_another)   # Add one list to another list
print(xs)
```

```
[3, 1, 5, 1, 3, 2, [10, 11], 100, 1000, 200]
```

Caution! "`extend`" vs "`append`" !

Lists documentation.

Slicing

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

```
nums = range(5)       # range is a built-in function that creates a list of integers
print( nums )         # In Python2 this would print an actual list, in Python3 it returns an iterator
```

```
range(0, 5)
```

Caution! Your Python2 code that has `range` may throw errors in Python3 interpreter!

```
nums = [0, 1, 2, 3, 4]
print( nums[2:4] )    # Get a slice from index 2 to 4 (exclusive)
print( nums[2:] )     # Get a slice from index 2 to the end
print( nums[:2] )     # Get a slice from the start to index 2 (exclusive)
print( nums[:] )      # Get a slice of the whole list
print( nums[:-1] )    # Slice indices can be negative
nums[2:4] = [8, 8, 23, 56, 9]  # Assign a new sublist to a slice
print( nums )
```

```
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 8, 23, 56, 9, 4]
```

Loops

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

```
animals = ['cat', 'dog', 'monkey']
i = 0
for animal in animals:
    i = i + 1
    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:

```
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:

```
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:

```
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:

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

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

IMPORTANT!

Dictionaries

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

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

```
cute
True
```

```python
d['fish'] = 'wet'    # Set an entry in a dictionary
print( d['fish'] )
```

```
wet
```

```python
print( d['monkey'] )
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-34-35510bf40422> in <module>()
----> 1 print( d['monkey'] )

KeyError: 'monkey'
```

SEARCH STACK OVERFLOW

```python
print( d.get('monkey', 'N/A') ) # Get an element with a default
print( d.get('fish', 'N/A') )   # Get an element with a default
```

```
N/A
wet
```

```python
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
```

Dictionaries documentation.

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

```python
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:

```python
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
```

If you want access to keys and their corresponding values (sorted by keys/values):

```python
d = {'a': 2, 'c': 4, 'd': 1, 'b': 10}
for animal, legs in sorted(d.items(), key = lambda x: x[0]):
    print( '%s - %d ' % (animal, legs) )

print ('')
for animal, legs in sorted(d.items(), key = lambda x: x[1]):
    print( '%s - %d ' % (animal, legs) )
```

```
a – 2
b – 10
c – 4
d – 1

d – 1
a – 2
c – 4
b – 10
```

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

```
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}
```

IMPORTANT!

Why do we need dictionary comprehension?

Sets

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

```
animals = { 'cat', 'dog', 'cat' }
print(animals)
```

```
{'cat', 'dog'}
```

```
animals = [ 'cat', 'dog', 'cat' ]
print(animals)
animals = set(animals)    # We can also use the 'set' function to create a set
print(animals)
```

```
['cat', 'dog', 'cat']
{'cat', 'dog'}
```

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

```
True
3
```

```
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:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print( '#%d: %s' % (idx + 1, animal) )
```

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

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

```
from math import sqrt
myset = {int(sqrt(x)) for x in range(30)}
print(myset)
```

```
{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:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
```

```
print (d)
t = (5, 6)        # Create a tuple
print( type(t) )
print( d[t] )
print( d[(1, 2)] )
```

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

```
t[0] = 1  # immutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-85a157a44726> in <module>()
----> 1 t[0] = 1  # immutable

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

SEARCH STACK OVERFLOW

Functions

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

```
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:

```
def hello(name, loud = False):
    if loud:
        print( 'HELLO, %s' % name.upper() )
    else:
        print( 'Hello, %s!' % name )
```

```
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-a75d7781aaeb> in <module>()
----> 1 hello()

TypeError: hello() missing 1 required positional argument: 'name'
```

SEARCH STACK OVERFLOW

```
hello('Fred')
```

```
Hello, Fred!
```

```
hello('Fred', True)
```

```
HELLO, FRED
```

What is wrong with the following function's arguments?
Note: Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.

```
def hello(loud = False, name):
    if loud:
        print( 'HELLO, %s' % name.upper() )
    else:
        print( 'Hello, %s!' % name )
```

```
  File "<ipython-input-54-2aa60dce0fbe>", line 1
    def hello(loud = False, name):
             ^
SyntaxError: non-default argument follows default argument
```

SEARCH STACK OVERFLOW

Classes

- All classes create objects, and all objects contain characteristics called **attributes**
- The **first argument of a method is `self`** - this is just a convention: the name self has absolutely no special meaning

Syntax for defining classes:

```
class A:
    def print_something(self):
        print( 'something' )

a = A()
a.print_something()
```

```
something
```

```
class Greeter:
    name = ""
    # Constructor (optional but needed for any initialization )
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud = False):
        if loud:
            print( 'HELLO' )
        else:
            print( 'Hello' )
```

```
g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method
g.greet(loud=True)   # Call an instance method
```

```
Hello
HELLO
```

Numpy

Numpy documentation.

- Numpy is the **core library** for scientific computing in Python
- It provides (a) a **high-performance** multidimensional array object, and (b) **tools** for working with these arrays.

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

```
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**.
- Numpy only supports numeric data types such as np.int8, np.int32, np.float32, np.half (float 16), etc.
- The number of dimensions is the **rank of the array**. For example an array of dimensions 2x3x4 has rank 3.
- 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. For example, `x = [[2,3][4,5]]`
- Elements can be accessed using square brackets. For example, `print( x[0] )`

Caution! Python arrays and Numpy arrays are different objects!
For example, `type` only works with Python data structures `shape` works with numpy arrays.

IMPORTANT!

```
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]
```

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

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

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

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

"**`shape`**" **is your hammer!**

Numpy also provides many functions to create arrays:

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

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

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

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

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

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

```python
d = np.eye(3)        # Create a 2x2 identity matrix
print(d)
```

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

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

```
[[0.18909388 0.85208646]
 [0.72517301 0.84151063]]
```

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

Here is an example matrix of **`shape (3, 4)`** (3 rows and 4 columns) and **`rank 2`** :

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

```python
import numpy as np

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 = 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.

IMPORTANT!

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

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

```
77
```

Then, how can we make a copy (not reference) of an array?

```
c = np.copy(a)
c[0, 0] = 100
print(c)
print(a)
```

```
[[100  77   3   4]
 [  5   6   7   8]
 [  9  10  11  12]]
[[ 1 77  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

You can also **mix integer indexing with slice indexing**. However, doing so will yield an array of lower rank than the original array.

```
# 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:

1. Mixing integer indexing with slices yields an array of lower rank.
2. Using only slices yields an array of the same rank as the original array.

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

```
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)
```

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

```
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing

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

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

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

# The returned array will have shape (3,)
print (' => ')
print( a[[0, 1, 2], [0, 1, 0]] )

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

```
a = 
[[1 2]
 [3 4]
 [5 6]]
 => 
[1 4]
 => 
[1 4 5]
 => 
[1 4 5]
```

```
# 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**:

```
# 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)
```

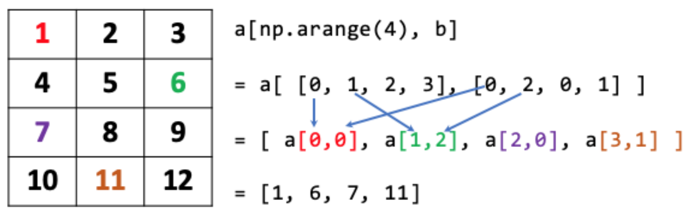
```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

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

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

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

```
[0 1 2 3]
[ 1  6  7 11]
```



```
a[np.arange(4), b]

= a[ [0, 1, 2, 3], [0, 2, 0, 1] ]

= [ a[0,0], a[1,2], a[2,0], a[3,1] ]

= [1, 6, 7, 11]
```

```
# 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:

```
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]]
```

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

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

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

Datatypes

- Every numpy array is a grid of elements **of the same type** (unlike a Python list)
- 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

```
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
```

Datatypes documentation.

Array math

- Basic mathematical functions operate elementwise on arrays
- These are available as both (a) operator overloads, and (b) functions in the numpy module

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

# Elementwise sum; both produce the array
print(x + y)         # operator overload
print(np.add(x, y))  # add available as function
```

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

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

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

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

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

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

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

```
# Elementwise square root; produces the array
print(np.sqrt(x))
```

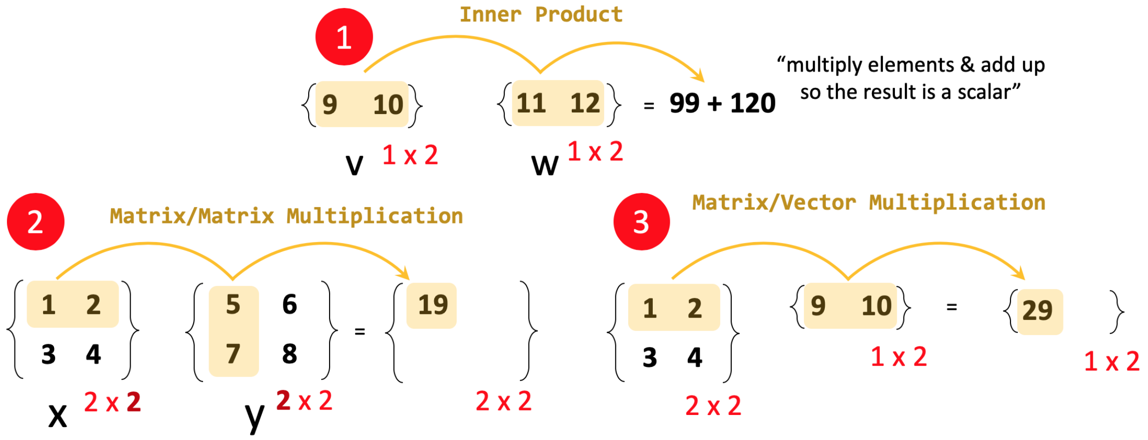
```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Elementwise Multiplication vs Dot Product:

- `*` is elementwise multiplication, **not matrix multiplication**
- 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

Dot Product

- `.dot()` function returns the dot product of two arrays
- For 1-D arrays, it is the inner product of the vectors
- For 2-D vectors, it is the equivalent to matrix multiplication
- If the dimensions don't match for matrix multiplication, it performs matrix/vector multiplication
- For N-dimensional arrays, it is a sum product over the last axis of a and the second-last axis of b



Examples:

```
v = np.array([9,10])
w = np.array([11, 12])
# Inner product of vectors
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

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

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

```
y = np.array([[5,6],[7,8]])
# Matrix / matrix product; both produce the rank 2 array
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`:

```
import numpy as np
x = np.array([[1,2],[3,4]])
print('x =')
print(x)
print('sum =')
print(np.sum(x))          # Compute sum of all elements
print('sum along axis 0 =')
print(np.sum(x, axis=0))  # Compute sum of each column
print('sum along axis 1 =')
print(np.sum(x, axis=1))  # Compute sum of each row
```

```
x =
[[1 2]
 [3 4]]
sum =
10
sum along axis 0 =
[4 6]
sum along axis 1 =
[3 7]
```

To transpose a matrix, simply use the T attribute of an array object:

```
print(x)
print(x.T)
```

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

Numpy arrays can also be reshaped using `.reshape()`

```
x = np.array([[1,2],[3,4]])
print(x.shape)

y = np.reshape(x, (1, 4))
```

```
print(y.shape)
print(y)
```

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

Mathematical functions documentation.

Introduction to 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**

Suppose that we want to add a constant vector to each row of a matrix:

```
# 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
print('x: ', x)
print('v: ', v)
```

```
x:  [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
v:  [1 0 1]
```

We would like to add `v` to `x`. Here is a naive way of doing so (**Method 1**):

```
# 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 (**Method 2**):

```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)
```

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

```
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** (**Method 3**):

```
import numpy as np

# We will add the vector v to each row of the matrix x,
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(x)
print('')
print(v)
print('')
print(y)
```

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

[1 0 1]

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

Broadcasting

- The term "broadcasting" describes how numpy treats arrays with different shapes during arithmetic operations
- The smaller array is "broadcast" across the larger array so that they have compatible shapes

The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```python
import numpy as np
a = np.array([1.0, 2.0, 3.0])
b = 2.0
print(a * b)
```

```
[2. 4. 6.]
```

Explanation:

- We can think of the **scalar b being stretched during the arithmetic operation** into an array with the same shape as a
- The new elements in b are simply copies of the original scalar
- The **stretching analogy is only conceptual**. NumPy is smart enough to use the original scalar value **without actually making copies**, so that broadcasting operations are **as memory and computationally efficient** as possible.

Broadcasting Rules

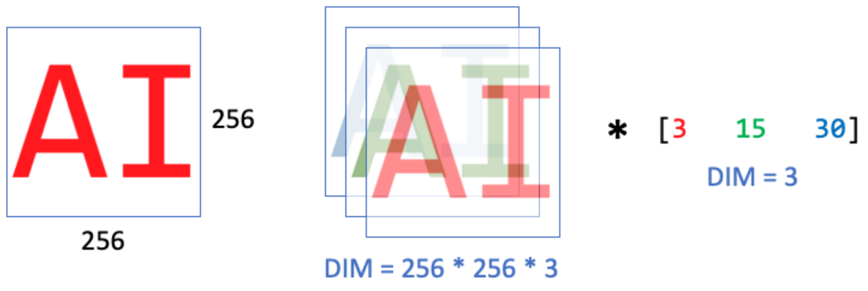
When operating on two arrays, NumPy **compares their shapes element-wise**. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible for broadcasting when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes. The **size of the resulting array is the maximum size along each dimension of the input arrays**.

Arrays do not need to have the same number of dimensions. For example, if you have a 256x256x3 array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```



When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or “copied” to match the other. Here, the scaling vector can be thought of having the dimensions `1 x 1 x 3`.

Broadcasting Examples

Examples where broadcasting works:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5

A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
```

```
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

Examples of shapes that do not broadcast:

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

Practice Example 1:

```python
x = np.arange(4)
y = np.ones(5)
print(x, x.shape)
print(y, y.shape)
```

```
[0 1 2 3] (4,)
[1. 1. 1. 1. 1.] (5,)
```

```python
# What will be the output?
print((x + y).shape)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-105-ba3d9005e40b> in <module>()
----> 1 print((x + y).shape)

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

SEARCH STACK OVERFLOW

Practice Example 2:

```python
xx = x.reshape(4,1)
print(xx, xx.shape)
print(y, y.shape)
```

```
[[0]
 [1]
 [2]
 [3]] (4, 1)
[1. 1. 1. 1. 1.] (5,)
```

```python
# What will be the output?
print((xx + y).shape)
```

```
(4, 5)
```

Outer Product using Broadcasting

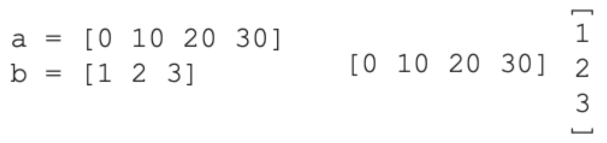
Broadcasting provides a convenient way of taking the outer product (or any other outer operation) of two arrays. The following **example shows an outer product operation** of two 1-d arrays:

```python
import numpy as np

a = np.array([0.0, 10.0, 20.0, 30.0])
b = np.array([1.0, 2.0, 3.0])

print(a, a.shape)
print(b, b.shape)
```

```
[ 0. 10. 20. 30.] (4,)
[1. 2. 3.] (3,)
```



```python
#Here the newaxis index operator inserts a new axis into a, making it a two-dimensional 4x1 array
print( a[:, np.newaxis] * b )
```

```
[[ 0.  0.  0.]
 [10. 20. 30.]
 [20. 40. 60.]
 [30. 60. 90.]]
```

Question. If we did not have np.newaxis feature, how would we compute outer product?

Functions that support broadcasting are known as universal functions. Here is the list.

Broadcasting documentation.

Matplotlib

- Matplotlib is a plotting library
- The `matplotlib.pyplot` module provides a plotting system

```
import matplotlib.pyplot as plt
import numpy as np
```

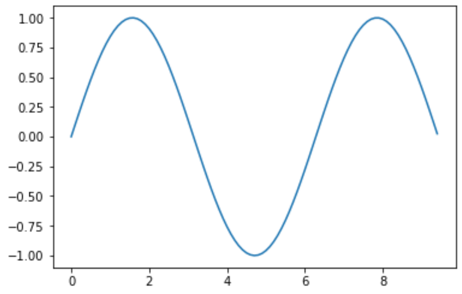
By running this special iPython command, we will be displaying plots inline:

```
%matplotlib inline
```

```
# 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)
```

```
[<matplotlib.lines.Line2D at 0x7fda28078c50>]
```

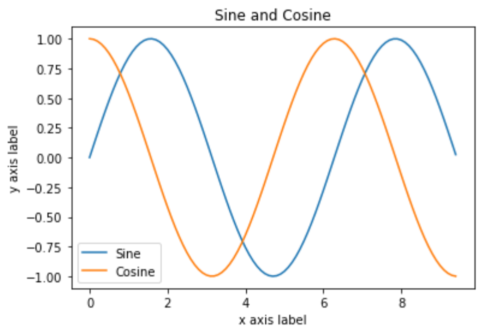


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

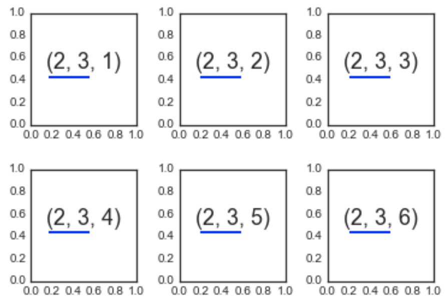
```
y_sin = np.sin(x)
y_cos = np.cos(x)

# 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'])
```

```
<matplotlib.legend.Legend at 0x7fda258161d0>
```



You can plot different things in the same figure using the subplot function.



Here is an example:

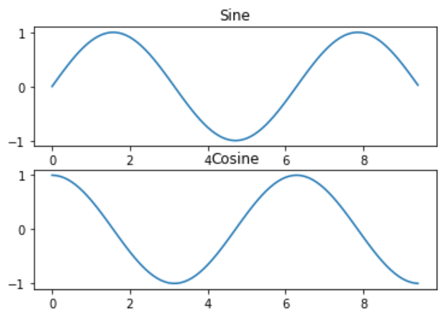
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# 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()
```



Subplot documentation.

Plotly

```
!pip install plotly_express
import plotly.express as px
```

```
Collecting plotly_express
  Downloading https://files.pythonhosted.org/packages/d4/d6/8a2906f51e073a4be80cab35cfa10e7a34853e60f3ed5304ac470852a08d/plotly_express-0.
Collecting plotly>=4.1.0 (from plotly_express)
  Downloading https://files.pythonhosted.org/packages/63/2b/4ca10995bfbdefd65c4238f9a2d3fde33705d18dd50914dd13302ec1daf1/plotly-4.1.0-py2.
     |████████████████████████████████| 7.1MB 2.8MB/s
Requirement already satisfied: scipy>=0.18 in /usr/local/lib/python3.6/dist-packages (from plotly_express) (1.3.1)
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.6/dist-packages (from plotly_express) (0.10.1)
Requirement already satisfied: pandas>=0.20.0 in /usr/local/lib/python3.6/dist-packages (from plotly_express) (0.24.2)
Requirement already satisfied: patsy>=0.5 in /usr/local/lib/python3.6/dist-packages (from plotly_express) (0.5.1)
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.6/dist-packages (from plotly_express) (1.16.4)
Requirement already satisfied: retrying>=1.3.3 in /usr/local/lib/python3.6/dist-packages (from plotly>=4.1.0->plotly_express) (1.3.3)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from plotly>=4.1.0->plotly_express) (1.12.0)
Requirement already satisfied: python-dateutil>=2.5.0 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.20.0->plotly_express) (2.5
Requirement already satisfied: pytz>=2011k in /usr/local/lib/python3.6/dist-packages (from pandas>=0.20.0->plotly_express) (2018.9)
Installing collected packages: plotly, plotly-express
  Found existing installation: plotly 3.6.1
    Uninstalling plotly-3.6.1:
      Successfully uninstalled plotly-3.6.1
Successfully installed plotly-4.1.0 plotly-express-0.4.1
```

```
iris = px.data.iris()
print(iris.shape)
iris.head()
```

```
(150, 6)
```

	sepal_length	sepal_width	petal_length	petal_width	species	species_id
0	5.1	3.5	1.4	0.2	setosa	1
1	4.9	3.0	1.4	0.2	setosa	1
2	4.7	3.2	1.3	0.2	setosa	1
3	4.6	3.1	1.5	0.2	setosa	1
4	5.0	3.6	1.4	0.2	setosa	1

```python
fig = px.scatter_3d(iris,
              x='sepal_length',
              y='sepal_width',
              z='petal_width',
              color='species')
fig.show()
```