09-numpy

December 5, 2019

Contents

1 NumPy

Sources: 1. This notebook is adapted from Numerical Computing in Python originally authored by Steve Phelps 2. Numerical Python: A Practical Techniques Approach for Industry book, Chapter 2: Vectors, matrices and multidimensional arrays

2 Basic information about numerical values

2.1 Scientific Notation in Python

- Python uses Scientific notation when it displays floating-point numbers:

[1]:
```
print(6720000000000000.0)
```

```
6720000000000000.0
```

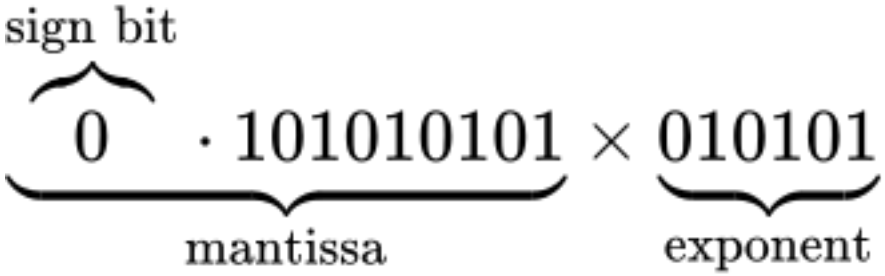
[2]:
```
print(67200000000000000.0)
```

```
6.72e+16
```

- Note that internally, the value is not *represented* exactly like this.
- Scientific notation is a convention for writing or rendering numbers, *not* representing them digitally.

2.2 Double and single precision formats

Numbers are stored as bits, and storing large numbers or floating point numbers needs a workaround. Here's mantissa and exponent



The number of bits allocated to represent each integer component of a float is given below:

Format	Sign	Exponent	Mantissa	Total
single	1	8	23	32
double	1	11	52	64

- Python normally works 64-bit precision.
- `numpy` allows us to specify the type when storing data in arrays.

- This is particularly useful for big data where we may need to be careful about the storage requirements of our data-set.

2.3 Not A Number (`NaN`) in Python

- Some mathematical operations on real numbers do not map onto real numbers.
- These results are represented using the special value to `NaN` which represents "not a (real) number".
- `NaN` is represented by an exponent of all 1s, and a non-zero mantissa.

[3]:
```
from numpy import sqrt, inf, isnan, nan
x = sqrt(-1)
x
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning:
invalid value encountered in sqrt
```

[3]:
```
nan
```

[4]:
```
y = inf - inf
y
```

[4]:
```
nan
```

2.4 Comparing `nan` values in Python

- Beware of comparing `nan` values

[5]:
```
x == y
```

[5]:
```
False
```

- To test whether a value is `nan` use the `isnan` function:

[6]:
```
isnan(x)
```

[6]:
```
True
```

2.5 `NaN` is not the same as `None`

- `None` represents a *missing* value.
- `NaN` represents an *invalid* floating-point value.
- These are fundamentally different entities:

[7]:
```
nan is None
```

[7]:
```
False
```

[8]:
```
isnan(None)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-8-4a0142ec4134> in <module>
----> 1 isnan(None)


TypeError: ufunc 'isnan' not supported for the input types, and the inputs␣
↪could not be safely coerced to any supported types according to the casting␣
↪rule ''safe''
```

2.6 Catastrophic Cancellation

- Suppose we have two real values x and $y = x + \epsilon$.
- ϵ is very small and x is very large.
- x has an *exact* floating point representation
- However, because of lack of precision x and y have the same floating point representation.
  - i.e. they are represented as the same sequence of 64-bits

[9]:
```
x = 3.141592653589793
x
```

[9]:
```
3.141592653589793
```

[10]:
```
y = 6.022e23
x = (x + y) - y
```

[11]:
```
x
```

[11]:
```
0.0
```

- Avoid subtracting two nearly-equal numbers.
- Especially in a loop!
- Better-yet use a well-validated existing implementation in the form of a numerical library.

3 Importing numpy

- Functions for numerical computiing are provided by a separate *module* called `numpy`.
- Before we use the numpy module we must import it.
- By convention, we import `numpy` using the alias `np`.
- Once we have done this we can prefix the functions in the numpy library using the prefix `np.`

[12]:
```
import numpy as np
```

- We can now use the functions defined in this package by prefixing them with `np`.

4 Arrays

- Arrays represent a collection of values.
- In contrast to lists:
  - arrays typically have a *fixed length*
    - they can be resized, but this involves an expensive copying process.
  - and all values in the array are of the *same type*.
    - typically we store floating-point values.
- Like lists:
  - arrays are *mutable*;
  - we can change the elements of an existing array.

4.1 Arrays in `numpy`

- Arrays are provided by the `numpy` module.
- The function `array()` creates an array given a list.

[13]:
```
import numpy as np
x = np.array([0, 1, 2, 3, 4])
x
```

[13]:
```
array([0, 1, 2, 3, 4])
```

[14]:
```
xlist = [0,1,2,3,4]
xlist
```

[14]:
```
[0, 1, 2, 3, 4]
```

4.2 Array indexing

- We can index an array just like a list

[15]:
```
x[4]
```

[15]:
```
4
```

[16]:
```
x[4] = 2
x
```

[16]:
```
array([0, 1, 2, 3, 2])
```

4.3 Arrays are not lists

- Although this looks a bit like a list of numbers, it is a fundamentally different type of value:

[17]:
```
type(x)
```

[17]:
```
numpy.ndarray
```

- For example, we cannot append to the array:

[18]:
```
x.append(5)
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-18-7e52d4acf950> in <module>
----> 1 x.append(5)


AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

5 Functions over arrays

- When we use arithmetic operators on arrays, we create a new array with the result of applying the operator to each element.

[19]:
```
y = x * 2
y
```

[19]:
```
array([0, 2, 4, 6, 4])
```

- The same goes for numerical functions:

[20]:
```
x = np.array([-1, 2, 3, -4])
y = abs(x)
```

```
y
```

[20]:
```
array([1, 2, 3, 4])
```

Remember, this is not possible with a list

[21]:
```
a_list=list(range(10))
a_list
```

[21]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[22]:
```
a_list * 2
```

[22]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[23]:
```
b_list= [-1,2,3,-4]
abs(b_list)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-23-5ad0e1699fa3> in <module>
  1 b_list= [-1,2,3,-4]
----> 2 abs(b_list)


TypeError: bad operand type for abs(): 'list'
```

[24]:
```
[ abs(num) for num in b_list ]
```

[24]:
```
[1, 2, 3, 4]
```

6 Vectorized functions

- Note that not every function automatically works with arrays.
- Functions that have been written to work with arrays of numbers are called *vectorized* functions.
- Most of the functions in `numpy` are already vectorized.
- You can create a vectorized version of any other function using the higher-order function `numpy.vectorize()`.

6.1 `vectorize` example

[25]:
```
def myfunc(x):
    if x >= 0.5:
        return x
    else:
        return 0.0

fv = np.vectorize(myfunc)
```

[26]:
```
x = np.arange(0, 1, 0.1)
x
```

[26]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

[27]:
```
fv(x)
```

[27]:
```
array([0. , 0. , 0. , 0. , 0. , 0.5, 0.6, 0.7, 0.8, 0.9])
```

[28]:
```
myfunc(x)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-28-c15edb3ac2c7> in <module>
----> 1 myfunc(x)


<ipython-input-25-3b7367a4f95b> in myfunc(x)
      1 def myfunc(x):
----> 2     if x >= 0.5:
      3         return x
      4     else:
      5         return 0.0


ValueError: The truth value of an array with more than one element is␣
↪ambiguous. Use a.any() or a.all()
```

7 Populating Arrays

- To populate an array with a range of values we use the `np.arange()` function:

[29]:
```
x = np.arange(0, 10)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

- We can also use floating point increments.

[30]:
```
x = np.arange(0, 1, 0.1)
print(x)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

[31]:
```
range(0,1,0.1)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-31-4ab091449f10> in <module>
----> 1 range(0,1,0.1)


TypeError: 'float' object cannot be interpreted as an integer
```

[33]:
```
[ i/10 for i in range(10) ]
```

[33]:
```
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

8 Plotting a sine curve

Let's make a list of numbers which are equivalent to 0, 30, 60, 90 and 120 degrees (multiples of pi/6).

[1]:
```
import math

thirties = [i*3.14/6 for i in range(0,5)]
thirties
```

[1]:
```
[0.0, 0.5233333333333333, 1.0466666666666666, 1.57, 2.0933333333333333]
```

Let's use `math.sin` for calculating sine value

Now, let's try `numpy.sin`

[76]:
```
import numpy as np
from numpy import pi, sin
```

```
sin(thirties)
```

[76]:
```
array([0.        , 0.4997701 , 0.86575984, 0.99999968, 0.8665558 ])
```

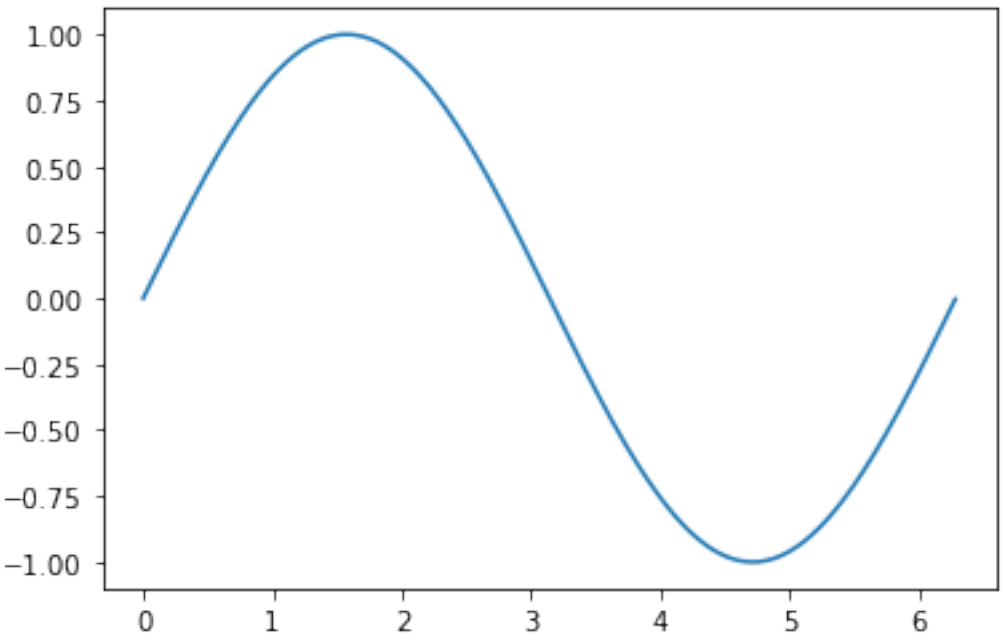
[3]:
```
import numpy

numpy.sin(thirties)
```

[3]:
```
array([0.        , 0.4997701 , 0.86575984, 0.99999968, 0.8665558 ])
```

math.sin is not vectorized, but numpy.sin is..

[10]:
```
%matplotlib inline

import numpy as np
from numpy import pi, sin
import matplotlib.pyplot as plt


x = np.arange(0, 2*pi, 0.01)
y = sin(x)
plt.plot(x, y)
```

[10]:
```
[<matplotlib.lines.Line2D at 0x7f091dcdc128>]
```



If we want to use `math.sin` we can not use it on array directly, list comprehension might be used to generate sine values and then used for plotting.

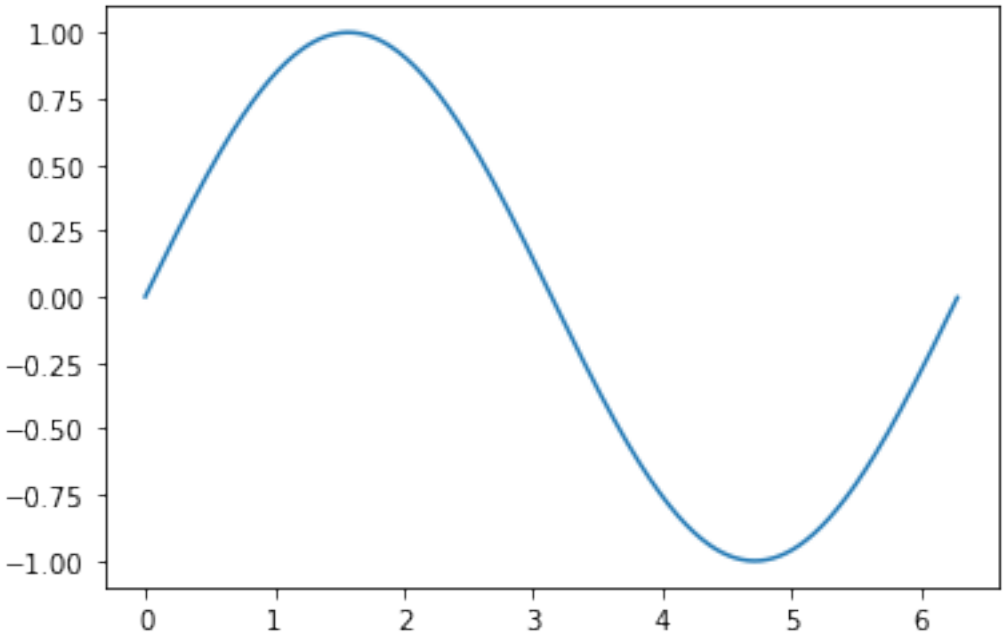
[78]:
```
import math

math.sin(x)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-78-73c024f15472> in <module>
  1 import math
  2
----> 3 math.sin(x)


TypeError: only size-1 arrays can be converted to Python scalars
```

[14]:
```
import math
y2 = [math.sin(i) for i in x]
```

[15]:
```
plt.plot(x,y2)
```

[15]:
```
[<matplotlib.lines.Line2D at 0x7f091d796240>]
```



Remember, last week we calculated `y` list for each `x` value in list via list comprehension (or by loop). Now, it's **vectorized**.

9 Multi-dimensional data

- Numpy arrays can hold multi-dimensional data.
- To create a multi-dimensional array, we can pass a list of lists to the `array()` function:

[1]:
```
import numpy as np

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

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

10 Arrays containing arrays

- A multi-dimensional array is an array of an arrays.
- The outer array holds the rows.
- Each row is itself an array:

[82]:
```
x[0]
```

[82]:
```
array([1, 2])
```

[83]:
```
x[1]
```

[83]:
```
array([3, 4])
```

- So the element in the second row, and first column is:

[84]:
```
x[1][0]
```

[84]:
```
3
```

11 Matrices

- We can create a matrix from a multi-dimensional array.

[2]:
```
M = np.matrix(x)
M
```

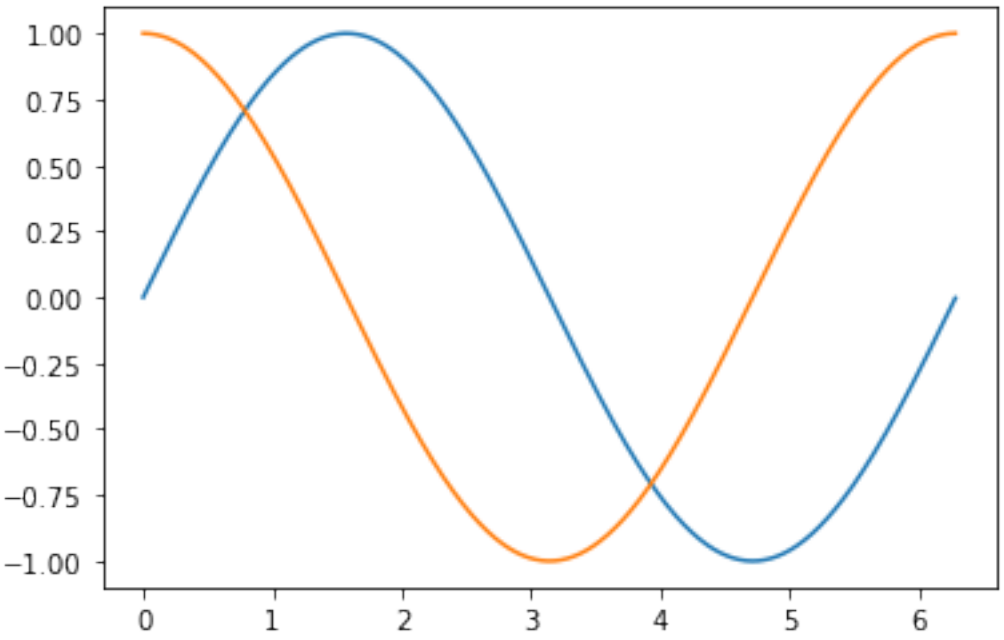
[2]:
```
matrix([[1, 2],
        [3, 4]])
```

12 Plotting multi-dimensional with matrices

- If we supply a matrix to `plot()` then it will plot the y-values taken from the *columns* of the matrix (notice the transpose in the example below).

[17]:
```
from numpy import pi, sin, cos
import matplotlib.pyplot as plt

x = np.arange(0, 2*pi, 0.01)
y = sin(x)
plt.plot(x, np.matrix([sin(x), cos(x)]).T)
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f091d7414a8>,
 <matplotlib.lines.Line2D at 0x7f091d7415f8>]
```



[87]:
```
np.matrix([sin(x), cos(x)])
```

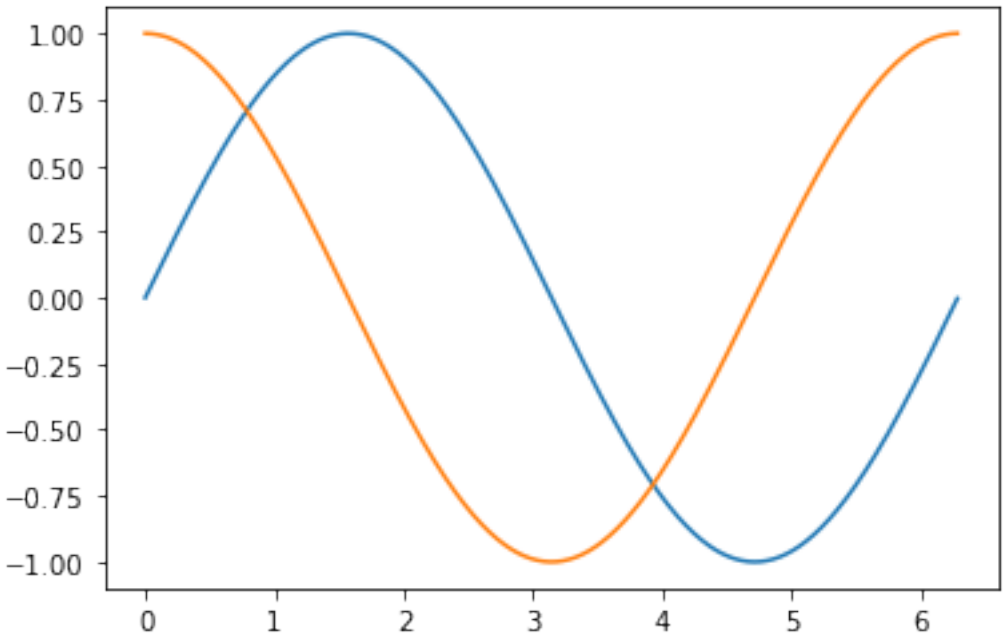
[87]:
```
matrix([[ 0.        ,  0.00999983,  0.01999867, ..., -0.02318323,
         -0.01318493, -0.0031853 ],
        [ 1.        ,  0.99995   ,  0.99980001, ...,  0.99973123,
          0.99991308,  0.99999493]])
```

[88]:
```
np.matrix([sin(x), cos(x)]).T
```

[88]:
```
matrix([[ 0.        ,  1.        ],
        [ 0.00999983,  0.99995   ],
        [ 0.01999867,  0.99980001],
        ...,
        [-0.02318323,  0.99973123],
        [-0.01318493,  0.99991308],
        [-0.0031853 ,  0.99999493]])
```

[27]:
```
x = np.arange(0, 2*pi, 0.01)
plt.plot(x, sin(x))
plt.plot(x, cos(x))
```

[27]:
```
[<matplotlib.lines.Line2D at 0x7f091d28b160>]
```



13 Performance

- When we use `numpy` matrices in Python the corresponding functions are linked with libraries written in C and FORTRAN.
- For example, see the BLAS (Basic Linear Algebra Subprograms) library.
- These libraries are very fast, and can be configured so that operations are performed in parallel on multiple CPU cores, or GPU hardware.

14 Matrix Operators

- Once we have a matrix, we can perform matrix computations.
- To compute the transpose and inverse use the `T` and `I` attributes:

To compute the transpose M^T

[89]:
```
M
```

[89]:
```
matrix([[1, 2],
        [3, 4]])
```

[90]:
```
M.T
```

[90]:
```
matrix([[1, 3],
        [2, 4]])
```

To compute the inverse M^{-1}

[91]:
```
M.I
```

[91]:
```
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Reminder

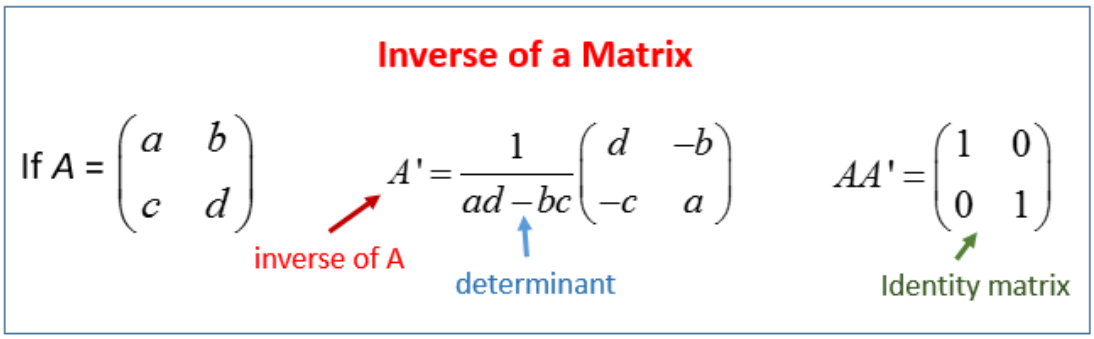


image source

15 Matrix Dimensions

- The total number of elements, and the dimensions of the array:

[92]:
```
M.size
```

[92]:
```
4
```

[93]:
```
M.shape
```

[93]:
```
(2, 2)
```

[94]:
```
len(M.shape)
```

[94]:
```
2
```

16 Creating Matrices from strings

- We can also create arrays directly from strings, which saves some typing:

[33]:
```
I2 = np.matrix('2 0; 0 2')
I2
```

[33]:
```
matrix([[2, 0],
        [0, 2]])
```

- The semicolon starts a new row.

17 Matrix addition

Two matrices must have an equal number of rows and columns to be added. The sum of two matrices A and B will be a matrix which has the same number of rows and columns as do A and B. The sum of A and B, denoted A + B, is computed by adding corresponding elements of A and B.

For example:

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

18 Matrix Multiplication

Now that we have two matrices, we can perform matrix multiplication:

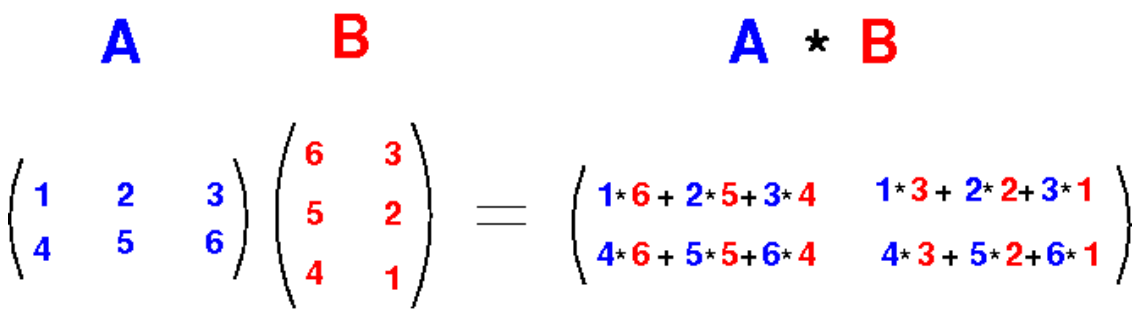


image source

[96]:
```
M * I2
```

[96]:
```
matrix([[2, 4],
        [6, 8]])
```

`dot` command can be used for matrix multiplication as well.

[97]:
```
np.dot(M, I2)
```

[97]:
```
matrix([[2, 4],
        [6, 8]])
```

18.0.1 Example

Let's practice matrix addition and multiplication with the matrix operations used in homework. Let's calculate this operation for x,y = 1,1

$$f_3(x,y) = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

[98]:
```
import numpy as np

left_leaf = np.matrix('0.20 -0.26; 0.23 0.22')
left_leaf
```

[98]:
```
matrix([[ 0.2 , -0.26],
        [ 0.23,  0.22]])
```

[99]:
```
coord = (1,1)
product = np.dot(left_leaf , np.matrix(coord).T)

product
```

```
[99]: matrix([[-0.06],
              [ 0.45]])
```

```
[100]: product + np.matrix((0.0,1.6)).T
```

```
[100]: matrix([[-0.06],
               [ 2.05]])
```

19 Matrix Indexing

- We can index and slice matrices using the same syntax as lists.

```
[101]: M
```

```
[101]: matrix([[1, 2],
               [3, 4]])
```

```
[102]: M[:,1]
```

```
[102]: matrix([[2],
               [4]])
```

20 Slices are references

- If we use this is an assignment, we create a *reference* to the sliced elements, *not* a copy.

```
[3]: V = M[:,1]  # This does not make a copy of the elements!
     V
```

```
[3]: matrix([[2],
             [4]])
```

```
[4]: M[0,1] = -2
     V
```

```
[4]: matrix([[-2],
             [ 4]])
```

Please visualize this at pythontutor page

Code to paste: (pythontutor does not support numpy)

```
M_list=[[2,3],[4,5]]

V = M_list[1]
print(V)
```

```
M_list[1][1] = -2
print(V)
```

20.1 Copying matrices and vectors

- To copy a matrix, or a slice of its elements, use the function `np.copy()`:

[5]:
```
M = np.matrix('1 2; 3 4')
V_copy = np.copy(M[:,1])  # This does copy the elements.
V_copy
```

[5]:
```
array([[2],
       [4]])
```

[6]:
```
M[0,1] = -2
V_copy
```

[6]:
```
array([[2],
       [4]])
```

[7]:
```
M
```

[7]:
```
matrix([[ 1, -2],
        [ 3,  4]])
```

21 Sums

One way we *could* sum a vector or matrix is to use a `for` loop.

[107]:
```
vector = np.arange(0.0, 100.0, 10.0)
vector
```

[107]:
```
array([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.])
```

[108]:
```
result = 0.0
for x in vector:
    result = result + x
result
```

[108]:
```
450.0
```

- This is not the most *efficient* way to compute a sum.

21.1 Efficient sums

- Instead of using a `for` loop, we can use a numpy function `sum()`.
- This function is written in the C language, and is very fast.

[109]:
```
vector = np.array([0, 1, 2, 3, 4])
print( np.sum(vector) )
```

```
10
```

21.2 Summing rows and columns

- When dealing with multi-dimensional data, the 'sum()' function has a named-argument `axis` which allows us to specify whether to sum along, each rows or columns.

[9]:
```
matrix = np.matrix('1 2 3; 4 5 6; 7 8 9')
print(matrix)
```

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

- To sum along columns:

[11]:
```
np.sum(matrix, axis=0)
```

[11]:
```
matrix([[12, 15, 18]])
```

- To sum along rows:

[112]:
```
np.sum(matrix, axis=1)
```

[112]:
```
matrix([[ 6],
        [15],
        [24]])
```

21.3 Cumulative sums

- Suppose we want to compute $y_n = \sum_{i=1}^{n} x_i$ where **x** is a vector.

[113]:
```
import numpy as np
x = np.array([0, 1, 2, 3, 4])
y = np.cumsum(x)
print(y)
```

```
[ 0  1  3  6 10]
```

21.4 Cumulative sums along rows and columns

[114]:
```
x = np.matrix('1 2 3; 4 5 6; 7 8 9')
print(x)
```

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

[115]:
```
y = np.cumsum(x)
np.cumsum(x, axis=0)
```

[115]:
```
matrix([[ 1,  2,  3],
        [ 5,  7,  9],
        [12, 15, 18]])
```

[116]:
```
np.cumsum(x, axis=1)
```

[116]:
```
matrix([[ 1,  3,  6],
        [ 4,  9, 15],
        [ 7, 15, 24]])
```

21.5 Cumulative products

- Similarly we can compute $y_n = \Pi_{i=1}^{n} x_i$ using `cumprod()`:

[12]:
```
import numpy as np
x = np.array([1, 2, 4, 3, 5])
np.cumprod(x)
```

[12]:
```
array([  1,   2,   8,  24, 120])
```

- We can compute cummulative products along rows and columns using the `axis` parameter, just as with the `cumsum()` example.

22 Generating (pseudo) random numbers

- The nested module `numpy.random` contains functions for generating random numbers from different probability distributions.

[14]:
```
from numpy.random import normal, uniform, exponential, randint
```

- Suppose that we have a random variable $\epsilon \sim N(0, 1)$.
- In Python we can draw from this distribution like so:

[23]:
```
epsilon = normal()
print(epsilon)
```

```
0.5546883473764372
```

- If we execute another call to the function, we will make a *new* draw from the distribution:

[10]:
```
epsilon = normal()
print(epsilon)
```

```
-1.7757792146371925
```

[47]:
```
for i in range(5):
    print(normal())
```

```
-0.3588289470012431
0.6034716026094954
-1.6647885294716944
-0.7001790376899514
1.1513910094871702
```

22.1 Pseudo-random numbers

- Strictly speaking, these are not random numbers.
- They rely on an initial state value called the *seed*.
- If we know the seed, then we can predict with total accuracy the rest of the sequence, given any "random" number.
- Nevertheless, statistically they behave like independently and identically-distributed values.
  - Statistical tests for correlation and auto-correlation give insignificant results.
- For this reason they called *pseudo*-random numbers.
- The algorithms for generating them are called Pseudo-Random Number Generators (PRNGs).
- Some applications, such as cryptography, require genuinely unpredictable sequences.
  - never use a standard PRNG for these applications!

22.2 Managing seed values

- In some applications we need to reliably reproduce the same sequence of pseudo-random numbers that were used.
- We can specify the seed value at the beginning of execution to achieve this.
- Use the function `seed()` in the `numpy.random` module.

22.3 Setting the seed

[28]:
```
from numpy.random import seed

seed(5)
```

[29]:
```
normal()
```

[29]:
```
0.44122748688504143
```

[30]:
```
normal()
```

[30]:
```
-0.33087015189408764
```

[50]:
```
seed(5)
for i in range(5):
    print(normal())
```

```
0.44122748688504143
-0.33087015189408764
2.43077118700778
-0.2520921296030769
0.10960984157818278
```

23 Drawing multiple variates

- To generate more than number, we can specify the `size` parameter:

[25]:
```
normal(size=10)
```

[25]:
```
array([ 2.43077119, -0.25209213,  0.10960984,  1.58248112, -0.9092324 ,
       -0.59163666,  0.18760323, -0.32986996, -1.19276461, -0.20487651])
```

- If you are generating very many variates, this will be *much* faster than using a for loop
- We can also specify more than one dimension:

[128]:
```
normal(size=(5,5))
```

[128]:
```
array([[-0.35882895,  0.6034716 , -1.66478853, -0.70017904,  1.15139101],
       [ 1.85733101, -1.51117956,  0.64484751, -0.98060789, -0.85685315],
       [-0.87187918, -0.42250793,  0.99643983,  0.71242127,  0.05914424],
       [-0.36331088,  0.00328884, -0.10593044,  0.79305332, -0.63157163],
       [-0.00619491, -0.10106761, -0.05230815,  0.24921766,  0.19766009]])
```

24 Array with specific type

You can pass in a second argument to array that gives the numeric type. There are a number of types listed here that your matrix can be. Some of these are aliased to single character codes. The most common ones are 'd' (double precision floating point number), 'D' (double precision complex number), and 'i' (int32).

[129]:
```
np.array([1,2,3,4,5,6])
```

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

```
[62]: np.array([1,2,3,4,5,6],'d')
```

```
[62]: array([1., 2., 3., 4., 5., 6.])
```

```
[131]: np.array([1,2,3,4,5,6],'D')
```

```
[131]: array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j, 6.+0.j])
```

```
[132]: np.array([1,2,3,4,5,6],'i')
```

```
[132]: array([1, 2, 3, 4, 5, 6], dtype=int32)
```

```
[59]: np.array([1,5, 'a', True])
```

```
[59]: array(['1', '5', 'a', 'True'], dtype='<U21')
```

25 Empty and identity matrices

`zeros` function is used for generating empty arrays or matrices

```
[133]: np.zeros((3,3),'d')
```

```
[133]: array([[0., 0., 0.],
              [0., 0., 0.],
              [0., 0., 0.]])
```

```
[134]: np.zeros((3,1))
```

```
[134]: array([[0.],
              [0.],
              [0.]])
```

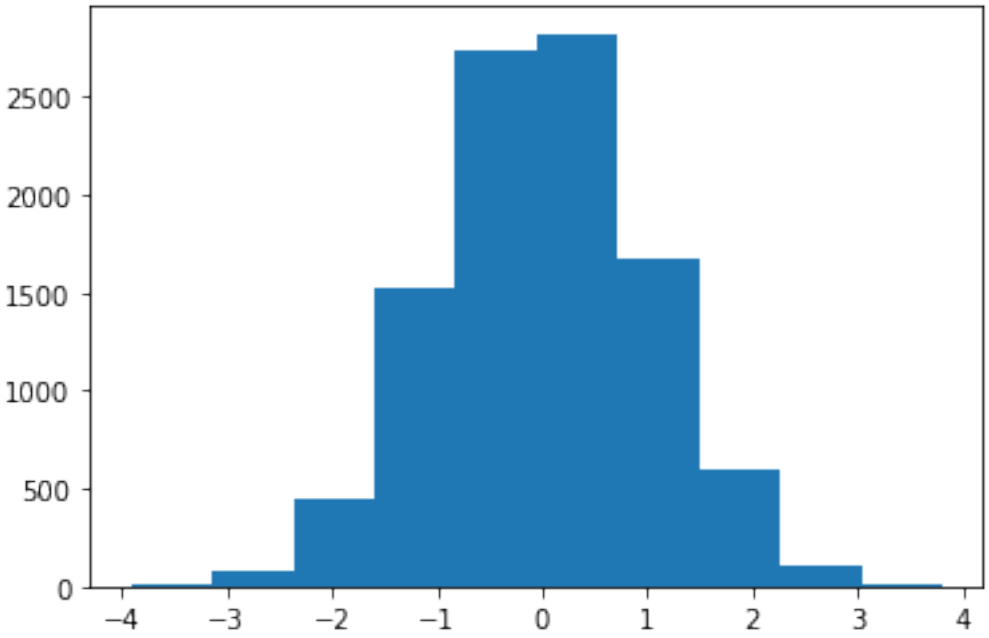
```
[135]: np.identity(4)
```

```
[135]: array([[1., 0., 0., 0.],
              [0., 1., 0., 0.],
              [0., 0., 1., 0.],
              [0., 0., 0., 1.]])
```

26 Histograms

- We can plot a histograms of randomly-distributed data using the `hist()` function from matplotlib:

[66]:
```
import matplotlib.pyplot as plt
%matplotlib inline

data = normal(size=10000)
#data[:10]
ax = plt.hist(data)
```



27 Computing histograms as matrices

- The function `histogram()` in the `numpy` module will count frequencies into bins and return the result as a 2-dimensional array.

[137]:
```
import numpy as np
np.histogram(data)
```

[137]:
```
(array([  23,  136,  618, 1597, 2626, 2635, 1620,  599,  130,   16]),
 array([-3.59780883, -2.87679609, -2.15578336, -1.43477063, -0.71375789,
         0.00725484,  0.72826758,  1.44928031,  2.17029304,  2.89130578,
         3.61231851]))
```

[67]:
```
(counts,boundaries)=np.histogram(data)
counts
```

[67]:
```
array([  12,   74,  447, 1528, 2737, 2822, 1670,  593,  105,   12])
```

[68]:
```
boundaries
```

[68]:
```
array([-3.91159411, -3.1399663 , -2.3683385 , -1.5967107 , -0.82508289,
       -0.05345509,  0.71817271,  1.48980051,  2.26142832,  3.03305612,
        3.80468392])
```

28 Summary statistics

- We can compute the summary statistics of a sample of values using the numpy functions `mean()` and `var()` to compute the sample mean $\bar{X}$ and sample variance σ^2_X .

[140]:
```
np.mean(data)
```

[140]:
```
-0.00045461080333497925
```

[141]:
```
np.var(data)
```

[141]:
```
1.0016048722546331
```

- These functions also have an `axis` parameter to compute mean and variances of columns or rows of a multi-dimensional data-set.

29 Summary statistics with `nan` values

- If the data contains `nan` values, then the summary statistics will also be `nan`.

[73]:
```
np.array([1,3,5,7, 0]) / 0
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning:
divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:1: RuntimeWarning:
invalid value encountered in true_divide
  """Entry point for launching an IPython kernel.
```

[73]:
```
array([inf, inf, inf, inf, nan])
```

[75]:
```
from numpy import nan
import numpy as np
data = np.array([1, 2, 3, 4, nan])
np.mean(data)
```

[75]:
```
nan
```

- To omit `nan` values from the calculation, use the functions `nanmean()` and `nanvar()`:

[76]:
```
np.nanmean(data)
```

[76]: 2.5

30 Discrete random numbers

- The `randint()` function in `numpy.random` can be used to draw from a uniform discrete probability distribution.
- It takes two parameters: the low value (inclusive), and the high value (exclusive).
- So to simulate one roll of a die, we would use the following Python code.

[77]:
```
die_roll = randint(0, 6) + 1
die_roll
```

[77]: 4

- Just as with the `normal()` function, we can generate an entire sequence of values.

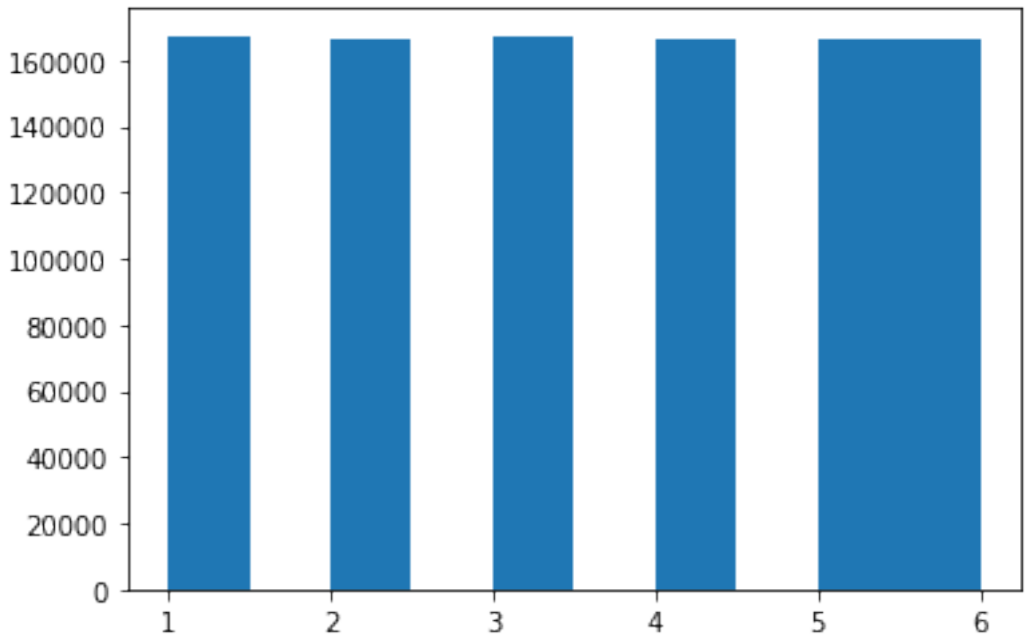
[78]:
```
trials = randint(0, 2, size = 20)
trials
```

[78]: array([0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0])

[88]:
```
many_dies = randint(1, 7, size = 1000000)
many_dies[:100]
```

[88]:
```
array([5, 1, 5, 6, 5, 1, 6, 6, 5, 5, 1, 1, 1, 4, 4, 4, 3, 6, 5, 1, 6, 5,
       5, 4, 3, 6, 5, 3, 5, 3, 3, 2, 6, 5, 4, 1, 1, 4, 6, 3, 2, 4, 3, 3,
       4, 5, 5, 2, 6, 5, 5, 5, 2, 3, 1, 1, 5, 4, 4, 5, 5, 3, 5, 2, 4, 2,
       6, 2, 2, 2, 6, 6, 3, 2, 4, 1, 1, 6, 2, 1, 3, 3, 3, 2, 2, 4, 2, 1,
       4, 5, 4, 5, 4, 4, 3, 3, 5, 6, 6, 2])
```

[89]:
```
ax = plt.hist(many_dies)
```



[90]:
```
import numpy as np
np.histogram(many_dies)
```

[90]:
```
(array([167421,      0, 166301,      0, 167110,      0, 166343,      0,
        166428, 166397]),
 array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ]))
```

31 Sequences, ranges

The linspace command makes a linear array of points from a starting to an ending value.

[ ]:
```
range vs. np.arange
```

[ ]:
```
0, 0.1, 0.2 .. , 1.0
```

[70]:
```
[ i/10 for i in range(0,11) ]
```

[70]:
```
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

[91]:
```
np.arange(0,1.1,0.1)
```

[91]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

[146]:
```
np.linspace(0,1)
```

```
[146]: array([0.        , 0.02040816, 0.04081633, 0.06122449, 0.08163265,
               0.10204082, 0.12244898, 0.14285714, 0.16326531, 0.18367347,
               0.20408163, 0.2244898 , 0.24489796, 0.26530612, 0.28571429,
               0.30612245, 0.32653061, 0.34693878, 0.36734694, 0.3877551 ,
               0.40816327, 0.42857143, 0.44897959, 0.46938776, 0.48979592,
               0.51020408, 0.53061224, 0.55102041, 0.57142857, 0.59183673,
               0.6122449 , 0.63265306, 0.65306122, 0.67346939, 0.69387755,
               0.71428571, 0.73469388, 0.75510204, 0.7755102 , 0.79591837,
               0.81632653, 0.83673469, 0.85714286, 0.87755102, 0.89795918,
               0.91836735, 0.93877551, 0.95918367, 0.97959184, 1.        ])
```

If you provide a third argument, it takes that as the number of points in the space. If you don't provide the argument, it gives a length 50 linear space.

```
[92]: np.linspace(0,1,11)
```

```
[92]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

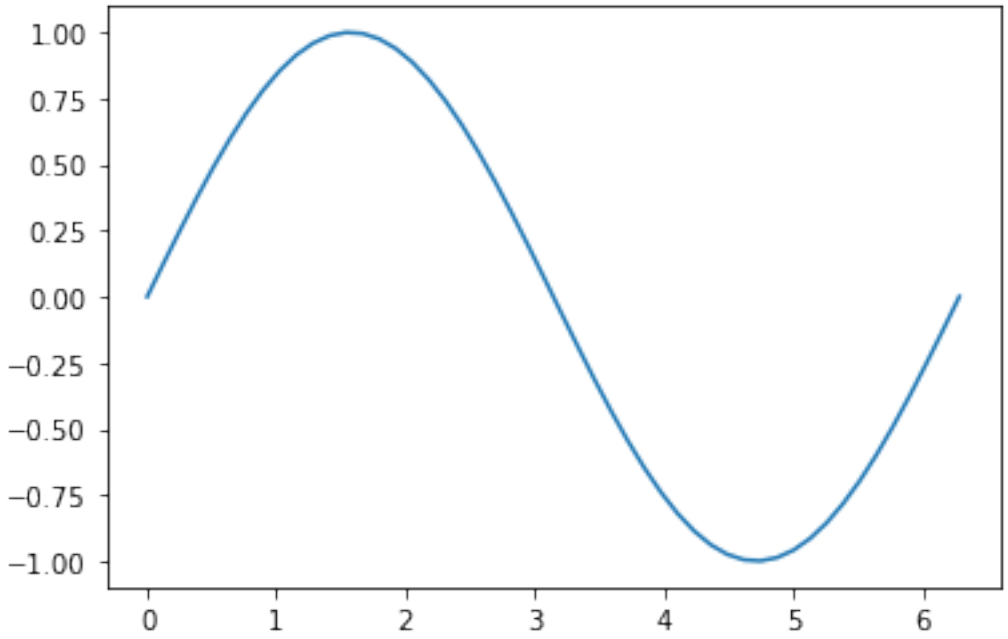
linspace is an easy way to make coordinates for plotting. Functions in the numpy library (all of which are imported into IPython notebook) can act on an entire vector (or even a matrix) of points at once. Thus,

```
[94]: from numpy import pi
      x = np.linspace(0,2*pi)
      np.sin(x)
```

```
[94]: array([ 0.00000000e+00,  1.27877162e-01,  2.53654584e-01,  3.75267005e-01,
              4.90717552e-01,  5.98110530e-01,  6.95682551e-01,  7.81831482e-01,
              8.55142763e-01,  9.14412623e-01,  9.58667853e-01,  9.87181783e-01,
              9.99486216e-01,  9.95379113e-01,  9.74927912e-01,  9.38468422e-01,
              8.86599306e-01,  8.20172255e-01,  7.40277997e-01,  6.48228395e-01,
              5.45534901e-01,  4.33883739e-01,  3.15108218e-01,  1.91158629e-01,
              6.40702200e-02, -6.40702200e-02, -1.91158629e-01, -3.15108218e-01,
             -4.33883739e-01, -5.45534901e-01, -6.48228395e-01, -7.40277997e-01,
             -8.20172255e-01, -8.86599306e-01, -9.38468422e-01, -9.74927912e-01,
             -9.95379113e-01, -9.99486216e-01, -9.87181783e-01, -9.58667853e-01,
             -9.14412623e-01, -8.55142763e-01, -7.81831482e-01, -6.95682551e-01,
             -5.98110530e-01, -4.90717552e-01, -3.75267005e-01, -2.53654584e-01,
             -1.27877162e-01, -2.44929360e-16])
```

```
[95]: import matplotlib.pyplot as plt
      import numpy as np
      from numpy import pi, sin
      x = np.linspace(0,2*pi)
      plt.plot(x,sin(x))
```

```
[95]: [<matplotlib.lines.Line2D at 0x7fe1a16f7c18>]
```



32 Acknowledgements

The earlier sections of this notebook were adapted from an article on floating-point numbers written by Steve Hollasch.

33 Finally

please follow the example at http://nbviewer.jupyter.org/url/atwallab.cshl.edu/teaching/QBbootcamp3.ipynb