1 Numpy

- A scientific computing library for Python
- Focuses on representation of data as **N-dimensional array**.
- Implements efficient computations and operations on these N-dimensional array. For example
  - Basic operations : +, - , * , /
  - Mathematical functions on these arrays : sin, cos, pow
  - Linear Algebra
  - Searching arrays, sorting arrays etc
- More efficient (than lists for ex) because
  - Specialized data structures that take advantage of homogenous typing, and contiguous storage
  - Many operations implemented in C
  - Some specialized methods are parallelized (kind of)
- Bottom line : When dealing with large data arrays, numpy arrays are a good data structure to use due to
  - Efficiency
  - Support for various operations (For example, how to sort an N-dimensional array?) including linear algebra operations, so saves you time writing all that code.
  - Shape and Broadcasting semantics (covered later)

To install numpy, run

```
pip install numpy
```

inside your environment

To import

```
import numpy
```

But usually everyone aliases it like this

In [1]:

```
import numpy as np
```

executed in 118ms, finished 18:09:24 2020-05-24

> **Resources** :
> The Official Numpy Quickstart Guide (https://numpy.org/devdocs/user/quickstart.html)
> A nice short tutorial on Python and Numpy (https://cs231n.github.io/python-numpy-tutorial)
> A cheatsheet covering most of the basic things you want to do with numpy (https://www.dataquest.io/blog/numpy-cheat-sheet/)

1.0.1 Numpy Basics

- np.array, attributes : size, shape, dtype, T,
- automatically creating some preset types of arrays

- zeros, ones, array, with type, np.arange, np.linspace
- np.random - rand, randn, randint

In [2]:

```
# np.array to create a numpy array from list

# you can create arrays with multiple types - int32, int64, float32, float64, bool
# interesting array attributes : size, shape, dtype, ndim, itemsize

x = np.array([1,2,3])
print(x, x.ndim, x.shape, x.size, x.dtype, x.itemsize)

x = np.array([1,2,3],dtype="int64")
print(x, x.ndim, x.shape, x.size, x.dtype,x.itemsize)

x = np.array([1,2,3],dtype="float32")
print(x, x.ndim, x.shape, x.size, x.dtype, x.itemsize)

x = np.array([1,2,3],dtype="float64")
print(x, x.ndim, x.shape, x.size, x.dtype,x.itemsize)

# otherwise numpy automatically guesses the type

x = np.array([1.5,2.6,7.7])
print(x, x.ndim, x.shape, x.size, x.dtype,x.itemsize)

x = np.array([True, False])
print(x, x.ndim, x.shape, x.size, x.dtype,x.itemsize)

# regular bool to int conversion
x = np.array([True, False],dtype="int32")
print(x, x.ndim, x.shape, x.size, x.dtype,x.itemsize)
```

executed in 19ms, finished 18:09:24 2020-05-24

```
[1 2 3] 1 (3,) 3 int64 8
[1 2 3] 1 (3,) 3 int64 8
[1. 2. 3.] 1 (3,) 3 float32 4
[1. 2. 3.] 1 (3,) 3 float64 8
[1.5 2.6 7.7] 1 (3,) 3 float64 8
[ True False] 1 (2,) 2 bool 1
[1 0] 1 (2,) 2 int32 4
```

In [3]:

```
# you can create multidimensional arrays

# a 2-d array

# sub arrays shapes have to be equal
x = np.array([[0.89012943, 0.95057931, 0.73957614],
       [0.32044759, 0.11317706, 0.21559414],
       [0.64161899, 0.22493727, 0.19792863]])

print(x.ndim, x.shape, x.size, x.dtype,x.itemsize)

# a 3-d array

x = np.array([[[0.95059078, 0.4920439 , 0.19623088],
        [0.35948661, 0.03963015, 0.01359072],
        [0.60065644, 0.43182609, 0.5839114 ]],

       [[0.49141504, 0.69129956, 0.53504228],
        [0.39380831, 0.90347317, 0.77581393],
        [0.77241933, 0.65354209, 0.00165317]],

       [[0.09852572, 0.48191009, 0.75421498],
        [0.73972572, 0.03630696, 0.88197898],
        [0.64376238, 0.77181998, 0.27395923]],

       [[0.86325292, 0.77815359, 0.85744606],
        [0.05330425, 0.79149102, 0.57572194],
        [0.92416049, 0.07126161, 0.15124192]]])

print(x.ndim, x.shape, x.size, x.dtype,x.itemsize)
```

executed in 7ms, finished 18:09:24 2020-05-24

```
2 (3, 3) 9 float64 8
3 (4, 3, 3) 36 float64 8
```

In [4]:

```
# numpy provides some functions to create some standard arrays
# np.zeros. np.ones, np.arange, np.linspace

print("*"*30)
# just zeros, you can specify the dtype
x=np.zeros(shape=(1,3,4),dtype="float32")
print(x)
print(x.ndim, x.shape, x.size, x.dtype,x.itemsize)


print("*"*30)
# just ones
x=np.ones((1,3,4))
print(x)
print(x.ndim, x.shape, x.size, x.dtype,x.itemsize)


print("*"*30)
# a range of integer (like python range())
x=np.arange(2,8,2)
print(x)
print(x.ndim, x.shape, x.size, x.dtype,x.itemsize)

print("*"*30)
# more generally to get equal divisions between two numbers
x=np.linspace(2.2,8.1,100)
print(x)
print(x.ndim, x.shape, x.size, x.dtype,x.itemsize)
```

executed in 13ms, finished 18:09:24 2020-05-24

```
******************************
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
3 (1, 3, 4) 12 float32 4
******************************
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
3 (1, 3, 4) 12 float64 8
******************************
[2 4 6]
1 (3,) 3 int64 8
******************************
[2.2        2.25959596 2.31919192 2.37878788 2.43838384 2.4979798
 2.55757576 2.61717172 2.67676768 2.73636364 2.7959596  2.85555556
 2.91515152 2.97474747 3.03434343 3.09393939 3.15353535 3.21313131
 3.27272727 3.33232323 3.39191919 3.45151515 3.51111111 3.57070707
 3.63030303 3.68989899 3.74949495 3.80909091 3.86868687 3.92828283
 3.98787879 4.04747475 4.10707071 4.16666667 4.22626263 4.28585859
 4.34545455 4.40505051 4.46464646 4.52424242 4.58383838 4.64343434
 4.7030303  4.76262626 4.82222222 4.88181818 4.94141414 5.0010101
 5.06060606 5.12020202 5.17979798 5.23939394 5.2989899  5.35858586
 5.41818182 5.47777778 5.53737374 5.5969697  5.65656566 5.71616162
 5.77575758 5.83535354 5.89494949 5.95454545 6.01414141 6.07373737
 6.13333333 6.19292929 6.25252525 6.31212121 6.37171717 6.43131313
 6.49090909 6.55050505 6.61010101 6.66969697 6.72929293 6.78888889
 6.84848485 6.90808081 6.96767677 7.02727273 7.08686869 7.14646465
 7.20606061 7.26565657 7.32525253 7.38484848 7.44444444 7.5040404
```

```
 7.56363636 7.62323232 7.68282828 7.74242424 7.8020202  7.86161616
 7.92121212 7.98080808 8.04040404 8.1        ]
1 (100,) 100 float64 8
```

In [5]:

```
# np.random is a module that gives options to create random arrays as well
# np.random.randint, np.random.uniform,

# a 1 dimensional array of random integers
x=np.random.randint(0,10, 20)
print(x, x.shape, x.dtype)
print("*"*30)

# any n-dimensional array of random numbers between 0 and 1 (uniformly distributed
x=np.random.random_sample(size=(3,2))
print(x, x.shape, x.dtype)
print("*"*30)

# more generally between a and b (uniformly distributed)
x=np.random.uniform(low=10,high=10.5,size=(3,2))
print(x, x.shape, x.dtype)
print("*"*30)
```

executed in 8ms, finished 18:09:24 2020-05-24

```
[8 6 8 9 6 4 6 8 9 9 9 2 8 0 1 9 0 9 0 1] (20,) int64
******************************
[[0.93974089 0.38253417]
 [0.69921738 0.48352944]
 [0.95757494 0.16032245]] (3, 2) float64
******************************
[[10.30940974 10.10595693]
 [10.11832505 10.19550529]
 [10.22452491 10.32324047]] (3, 2) float64
******************************
```

- slicing and indexing, all indices vs few indices

In [6]:

```
# numpy indexing and slicing
# negative indices
# complete vs incomplete slices : x[0] vs x[0,:,:]
# mixing indices and slices

x=np.random.uniform(low=10,high=10.5,size=(4,6,5))

print(x[0].shape)
print(x[1:3].shape)
print(x[1:-1].shape)
print(x[1:-1,:,:].shape)
print(x[1:3,2:5,3:].shape)
print(x[1:3,2,3:].shape)
```

executed in 6ms, finished 18:09:24 2020-05-24

```
(6, 5)
(2, 6, 5)
(2, 6, 5)
(2, 6, 5)
(2, 3, 2)
(2, 2)
```

- operations on arrays of same size
  - numpy mathematical functions - sin, cos, log etc
  - scalar arithmetic
  - arithmetic operations, upcasting
  - comparison operations
    - np.all, np.any
    - np.where, np.argwhere, np.nonzero

In [7]:

```
x=np.random.randint(low=1,high=10,size=(2,3))
print(x)
```

executed in 4ms, finished 18:09:24 2020-05-24

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

In [8]:

```
# You can do arithmetic between numpy arrays and scalars

print(x,end="\n\n")
print(x+2,end="\n\n")
print(x-2,end="\n\n")
print(x*2,end="\n\n")
print(x/2,end="\n\n")
print(x**2,end="\n\n")
```

executed in 6ms, finished 18:09:24 2020-05-24

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

[[ 4  7  8]
 [11  9  6]]

[[0 3 4]
 [7 5 2]]

[[ 4 10 12]
 [18 14  8]]

[[1.  2.5 3. ]
 [4.5 3.5 2. ]]

[[ 4 25 36]
 [81 49 16]]
```

In [9]:

```
# You can use some elementwise global functions on numpy arrays
# like np.sin, cos, exp, log, sqrt

print(np.sin(x),end="\n\n")
print(np.cos(x),end="\n\n")
print(np.sin(x)**2 + np.cos(x)**2,end="\n\n")
print(np.sqrt(x),end="\n\n")
```

executed in 5ms, finished 18:09:24 2020-05-24

```
[[ 0.90929743 -0.95892427 -0.2794155 ]
 [ 0.41211849  0.6569866  -0.7568025 ]]

[[-0.41614684  0.28366219  0.96017029]
 [-0.91113026  0.75390225 -0.65364362]]

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

[[1.41421356 2.23606798 2.44948974]
 [3.         2.64575131 2.        ]]
```

In [10]:

```
x=np.random.randint(low=1,high=10,size=(2,3),dtype="int32")
print(x)
y=np.random.randint(low=1,high=10,size=(2,3),dtype="int64")
print(y)
```

executed in 5ms, finished 18:09:24 2020-05-24

```
[[7 4 1]
 [7 1 2]]
[[3 5 3]
 [5 1 4]]
```

In [11]:

```
# elementwise binary arithmetic operations on arrays of same shape
# + , - , *, /
# upcasting
print(x+y,end="\n\n")
print(x-y,end="\n\n")
print(x*y,end="\n\n")
print(x/y,end="\n\n")
print(x**y,end="\n\n")

# Operations between different types results in upcasting to a compatible larger t
# which can contain both types : ex : int32 + float32 != float32
print(x.dtype)
zeros=np.zeros((2,3),dtype="float32")
print((x+zeros).dtype)
```

executed in 8ms, finished 18:09:24 2020-05-24

```
[[10  9  4]
 [12  2  6]]

[[ 4 -1 -2]
 [ 2  0 -2]]

[[21 20  3]
 [35  1  8]]

[[2.33333333 0.8        0.33333333]
 [1.4        1.         0.5        ]]

[[  343  1024     1]
 [16807     1    16]]

int32
float64
```

In [12]:

```
# You can also compare numpy arrays with other arrays or scalars
# For now we restrict to both arrays having the same shape
# elementwise comparison operators >, <, ==, != etc
print(x > 2)
print(x > y)
```

executed in 3ms, finished 18:09:24 2020-05-24

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

In [13]:

```
# np.all(boolean_array) is True if all of the elements in boolean_array is True
# np.any(boolean_array) is True if at least one of the elements in boolean_array i
print(np.all(x>0))
print(np.any(x==3))
```

executed in 4ms, finished 18:09:24 2020-05-24

```
True
False
```

In [14]:

```
# np.where(boolean_array): Get tuple of indices where boolean array is True
print(x)
print("\n")
print(y)
print("\n")
print(np.where(x>y))
print("\n")
# np.where(boolean_array, A, B): If entry in boolean_array is True, choose entry f
# Thus, A,B, and boolean_array have to be of the same shape (or A or B can be scal
max_xy=np.maximum(x,y)
max_xy_w=np.where(x>y,x,y)
print(np.all(max_xy_w==max_xy))
print("\n")
print(np.where(x>3,20,-20))
```

executed in 6ms, finished 18:09:24 2020-05-24

```
[[7 4 1]
 [7 1 2]]


[[3 5 3]
 [5 1 4]]


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


True


[[ 20  20 -20]
 [ 20 -20 -20]]
```

In [15]:

```
# np.nonzero(boolean_array) : Get tuple of indices where boolean array is True, sa
# np.argwhere(boolean_array) : Something like a transpose of np.nonzero...
print(x)
print(np.nonzero(x>2))
print(np.argwhere(x>2))
```

executed in 4ms, finished 18:09:24 2020-05-24

```
[[7 4 1]
 [7 1 2]]
(array([0, 0, 1]), array([0, 1, 0]))
[[0 0]
 [0 1]
 [1 0]]
```

Miscellaneous

- astype()
- sum, min, max, argmin, argmax, sort, argsort
- tolist()
- copy()
- np.concatenate

In [16]:

```
# arr.astype()
x=np.random.randint(low=1,high=10,size=(2,3),dtype="int32")
print(x.dtype)
x=x.astype("float32")
print(x.dtype)
x=x.astype("bool")
print(x.dtype)
```

executed in 5ms, finished 18:09:24 2020-05-24

```
int32
float32
bool
```

In [17]:

```
# np.sort, np.max, np.argsort, np.argmax w/ and w/o axis parameter
x=np.random.randint(low=1,high=10,size=(4,3),dtype="int32")
print(x)
print(np.min(x))
print(np.min(x,axis=0))
print(np.min(x,axis=1))
print("\n"+"*"*10+"\n")
print(np.argmax(x))
print(np.argmax(x,axis=0))
print(np.argmax(x,axis=1))

# np sort : default is -1

print("\n"+"*"*10+"\n")
print(np.sort(x))
print(np.sort(x,axis=0))
print(np.sort(x,axis=1))

print("\n"+"*"*10+"\n")
print(np.argsort(x))
print(np.argsort(x,axis=0))
print(np.argsort(x,axis=1))
```

executed in 10ms, finished 18:09:24 2020-05-24

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

**********

10
[1 3 2]
[1 1 2 1]

**********

[[1 4 4]
 [2 5 7]
 [3 4 7]
 [3 4 9]]
[[1 4 2]
 [3 4 3]
 [4 7 4]
 [5 9 7]]
[[1 4 4]
 [2 5 7]
 [3 4 7]
 [3 4 9]]

**********

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

```
[[0 0 1]
 [2 2 3]
 [3 1 0]
 [1 3 2]]
[[0 1 2]
 [2 0 1]
 [0 1 2]
 [2 0 1]]
```

In [18]:

```
# arr.tolist()

x.tolist()
```

executed in 10ms, finished 18:09:24 2020-05-24

Out[18]:

```
[[1, 4, 4], [5, 7, 2], [3, 4, 7], [4, 9, 3]]
```

1.0.2 Broadcasting

- Can you perform `A+B` when `a.shape != b.shape` ?
- Sometimes, when their shapes are **compatible**

How do you decide compatibility?

- Case 1 : ndim is equal
- Case 2 : ndim is not equal

- For Case 1, for every axis where shape doesn't match, **one of the dimensions has to be 1**.
- In that case the array with dimension = 1 is "tiled" or "duplicated" along the non matching axis, to match the larger array.
- For example : `A of shape (4,1,5) + B of shape(4,5,5)` , A is copied 5 times along the `1` axis to match the shape with B. Then it can be added.
- Similarly, for the case `A of shape(1,4,5) + B of shape(3,4,1)`
  - A is copied along the 0-axis 3 times
  - B is copied along the 2-axis 5 times
  - Thereafter we can add A + B

- For Case 2, we first make ndim equal for both arrays - by prepending the smaller array with 1s.
- For example, `A of shape (4,4,5,5) + B of shape(5,5)` , then first we convert B to shape `(1,1,5,5)` .
- Thereafter we apply Case 1. In this case A and B are broadcastable.
- Another example, `A of shape (2,5) + B of shape(2,4,5)` , then first we convert A to shape `(1,2,5)` . Then, in Case 1 however broadcasting will fail, because the 1-dimension is not same (2 vs 4).

So which of the following combinations work for broadcasting?

- (3,4,5) vs (4,5) vs (1,4,5)
- (3,4,5) vs (5,) vs (1,5) vs (1,1,5)
- (3,4,5) vs (1,) vs (1,1) vs (1,1,1) : Scalar case!
- (3,4,5) vs (3,1,5)
- (3,4,5) vs (4,4,5) ?
- (3,4,5) vs (3,8,5) ?
- (3,4,5) vs (3,4,7) ?

https://numpy.org/devdocs/user/basics.broadcasting.html
(https://numpy.org/devdocs/user/basics.broadcasting.html)

In [19]:

```
# Let's try the different cases we mentioned above and see what happens
x=np.random.randint(low=1,high=10,size=(2,3,2),dtype="int32")
y=np.random.randint(low=1,high=10,size=(2,3,2),dtype="int32")
print(x)
print(y)
print(x+y)
```

executed in 5ms, finished 18:09:24 2020-05-24

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

 [[6 3]
  [1 5]
  [4 1]]]
[[[6 6]
  [8 1]
  [5 9]]

 [[5 5]
  [5 9]
  [4 1]]]
[[[ 9  8]
  [10  5]
  [ 7 10]]

 [[11  8]
  [ 6 14]
  [ 8  2]]]
```

1.0.3 Manipulating shape

- expand_dims, squeeze, flatten, reshape

In [20]:

```
x= np.random.random((5,5))
print(x)
```

executed in 4ms, finished 18:09:24 2020-05-24

```
[[0.65628936 0.53580925 0.61111055 0.55807366 0.17518485]
 [0.89396368 0.51182689 0.48569536 0.08317149 0.80857819]
 [0.23172367 0.51990505 0.64523944 0.56416134 0.3608993 ]
 [0.60397349 0.30386437 0.22384056 0.23004792 0.7280163 ]
 [0.29966579 0.76987699 0.62173972 0.70876911 0.66961209]]
```

In [21]:

```
# flatten()
x.flatten()
```

executed in 4ms, finished 18:09:24 2020-05-24

Out[21]:

```
array([0.65628936, 0.53580925, 0.61111055, 0.55807366, 0.17518485,
       0.89396368, 0.51182689, 0.48569536, 0.08317149, 0.80857819,
       0.23172367, 0.51990505, 0.64523944, 0.56416134, 0.3608993 ,
       0.60397349, 0.30386437, 0.22384056, 0.23004792, 0.7280163 ,
       0.29966579, 0.76987699, 0.62173972, 0.70876911, 0.66961209])
```

In [22]:

```
# expand_dims(a,axis)
# add a singleton dimension
x2=np.expand_dims(x,axis=0)
print(x2.shape)
x2=np.expand_dims(x,axis=1)
print(x2.shape)
x2=np.expand_dims(x,axis=2)
print(x2.shape)
x2=np.expand_dims(x,axis=-1)
print(x2.shape)
```

executed in 5ms, finished 18:09:24 2020-05-24

```
(1, 5, 5)
(5, 1, 5)
(5, 5, 1)
(5, 5, 1)
```

In [23]:

```
# squeeze(a,axis=1 or (1,2) or None)
# remove (some/all) singleton dimension
x2.squeeze().shape
```

executed in 4ms, finished 18:09:24 2020-05-24

Out[23]:

```
(5, 5)
```

In [24]:

```
# moveaxis(a,[0,1,2],[1,2,0])
# 0 ->1 , 1 -> 2 , 2 ->0  (a cycle)
print(x2.shape)
x3=np.moveaxis(x2,[0,1,2],[1,2,0])
print(x3.shape)

print(x,x.shape)
x3=np.moveaxis(x,[0,1],[1,0])
print(x3,x3.shape)
```

executed in 6ms, finished 18:09:24 2020-05-24

```
(5, 5, 1)
(1, 5, 5)
[[0.65628936 0.53580925 0.61111055 0.55807366 0.17518485]
 [0.89396368 0.51182689 0.48569536 0.08317149 0.80857819]
 [0.23172367 0.51990505 0.64523944 0.56416134 0.3608993 ]
 [0.60397349 0.30386437 0.22384056 0.23004792 0.7280163 ]
 [0.29966579 0.76987699 0.62173972 0.70876911 0.66961209]] (5, 5)
[[0.65628936 0.89396368 0.23172367 0.60397349 0.29966579]
 [0.53580925 0.51182689 0.51990505 0.30386437 0.76987699]
 [0.61111055 0.48569536 0.64523944 0.22384056 0.62173972]
 [0.55807366 0.08317149 0.56416134 0.23004792 0.70876911]
 [0.17518485 0.80857819 0.3608993  0.7280163  0.66961209]] (5, 5)
```

In [25]:

```
# reshape
# note : this method conceptually first flattens the array
# and then fills it into the new array in row first order
# while reshape can do the work of flatten, squeeze and expand_dims,
# reshape CANNOT be used to moveaxis - even though
# trying will not give an error - don't fall into this trap!

a=np.arange(1,25,1)
print(a.reshape(4,6))
print(a.reshape(6,4))
print(a.reshape(2,3,4))
```

executed in 4ms, finished 18:09:24 2020-05-24

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]]
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]

 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]]
```

1.0.4 Further Reading

- np.vstack,np.hstack,np.tile
- integer indexing and boolean indexing
- cumsum, cumprod,
- serializing arrays : np.save

2 Matplotlib

> Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

- In simple words - a library to plot data in various ways.
- With matplotlib you can
  - make line plots, scatter plots, pie charts, bar charts, matrix visualization etc
  - plot in 2d as well as 3d
  - draw and display arbitrary images in python
- matplotlib works with numpy arrays - i.e. the data to plot is numpy arrays.
- Generally everything is done using the `matplotlib.pyplot` module - which has most of the functions that we will need.

To install matplotlib run the following command in your python environment

```
pip install matplotlib
```

To import matplotlib, we will be rather importing the submodule pyplt

In [26]:

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

executed in 271ms, finished 18:09:24 2020-05-24

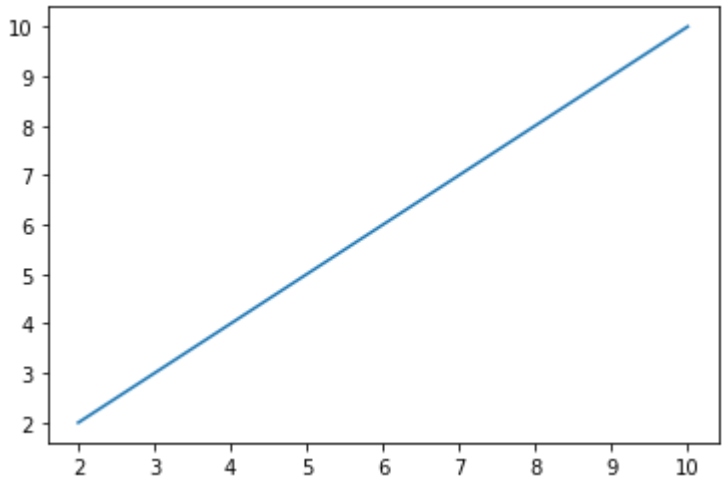
> **Resources**
>
> - An overview of matplotlib (https://matplotlib.org/tutorials/introductory/usage.html)
>   The above resource is advanced, but you can read the sections
>   A simple example, Parts of a Figure, Backends : What is a backend?
> - The pyplot tutorial (https://matplotlib.org/tutorials/introductory/pyplot.html)
>   Definitely read through the pyplot tutorial
> - A list of all the functions in pyplt (https://matplotlib.org/api/pyplot_summary.html)
>   You should definitely go through this list and check out 10-15 functions to get a feel for what is possible with matplotlib
> - The gallery of examples (https://matplotlib.org/gallery/index.html)

2.0.1 Some simple plots

In [27]:

```
# plot y=x
num_samples=10
x=np.linspace(2,10,num_samples)
y=x
_=plt.plot(x,y)
```

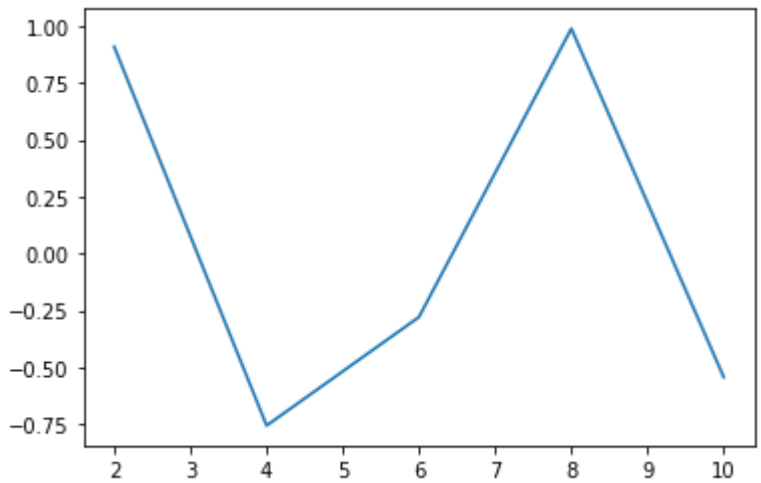
executed in 178ms, finished 18:09:25 2020-05-24



In [28]:

```
# plot y=sin(x)
num_samples=5
x=np.linspace(2,10,num_samples)
y=np.sin(x)
_=plt.plot(x,y)
```

executed in 179ms, finished 18:09:25 2020-05-24

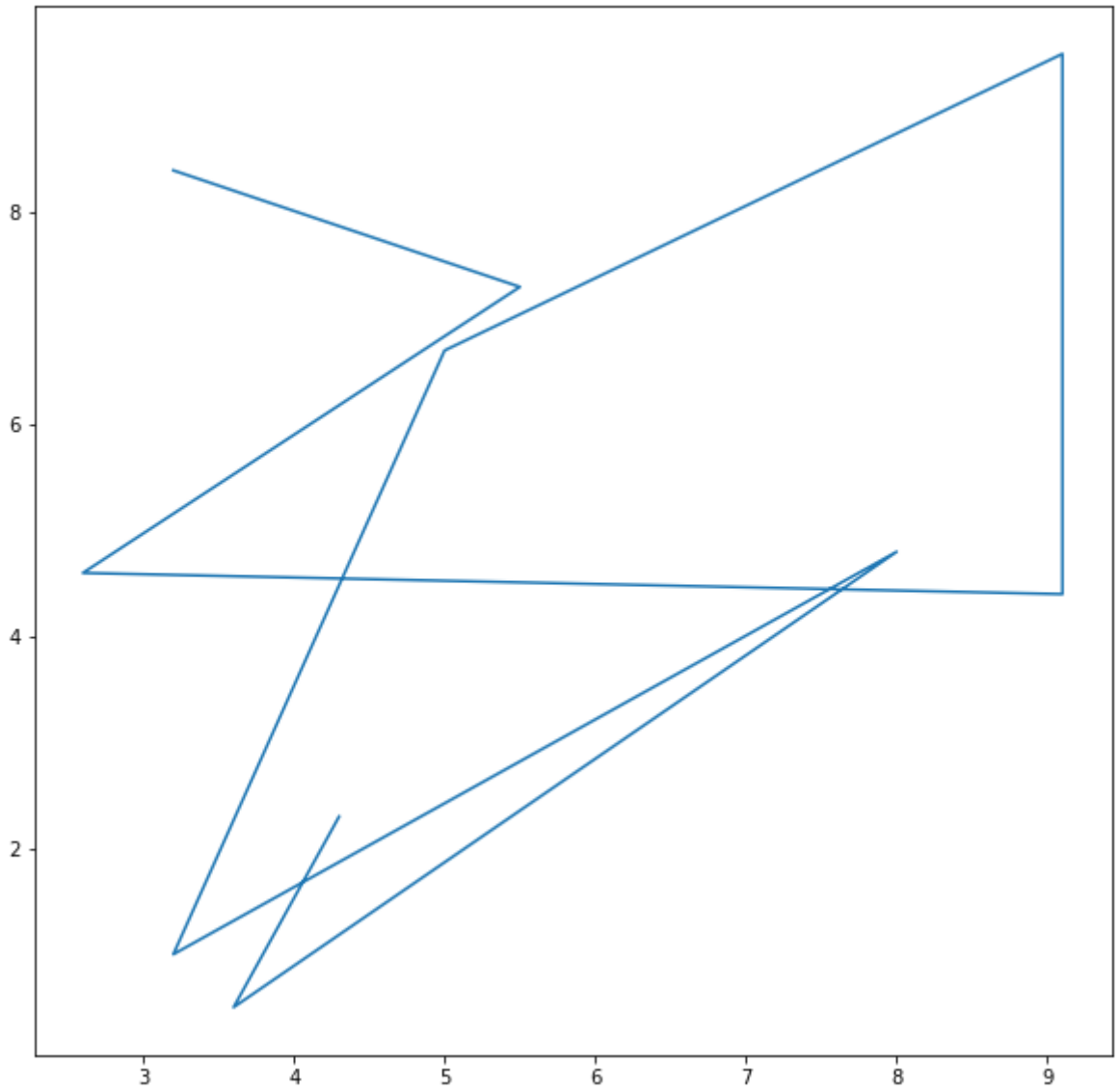


In [29]:

```
# plot arbitrary lines
x=np.round(np.random.uniform(0,10,10),decimals=1)
y=np.round(np.random.uniform(0,10,10),decimals=1)
print(list(zip(x,y)))
plt.figure(figsize=(10,10))
_=plt.plot(x,y)
```

executed in 233ms, finished 18:09:25 2020-05-24

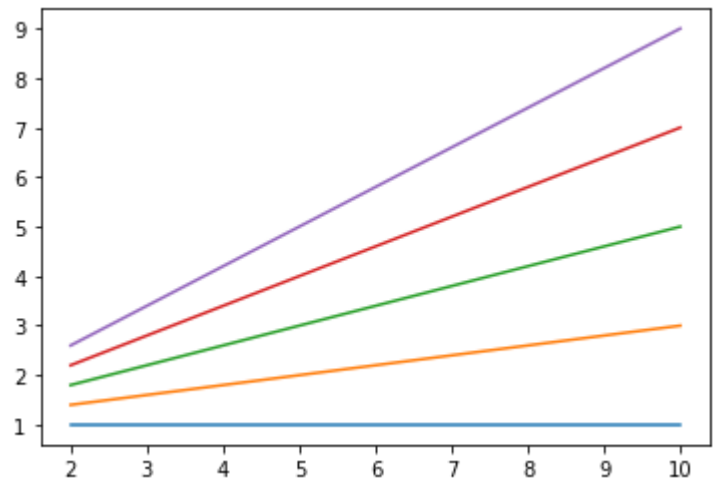
```
[(4.3, 2.3), (3.6, 0.5), (8.0, 4.8), (3.2, 1.0), (5.0, 6.7), (9.1, 9.
5), (9.1, 4.4), (2.6, 4.6), (5.5, 7.3), (3.2, 8.4)]
```



In [30]:

```
# plot multiple lines

num_samples=5
num_lines=5
for i in range(0,num_lines):
    x=np.linspace(2,10,num_samples)
    y=(i/num_lines)*x + 1
    _=plt.plot(x,y)
```

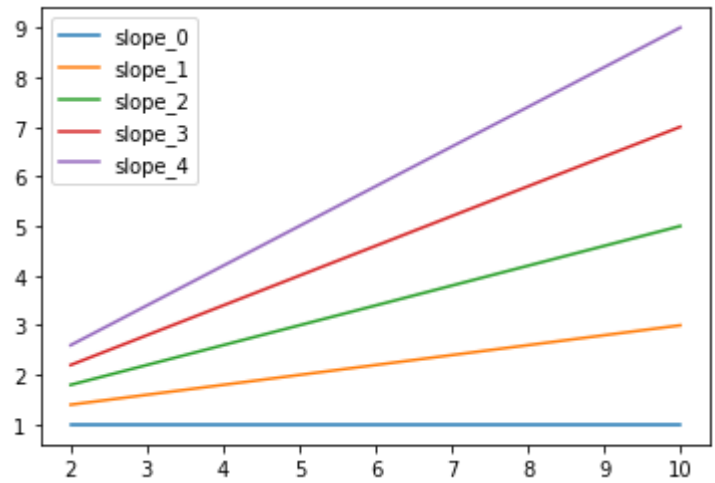
executed in 150ms, finished 18:09:25 2020-05-24



In [31]:

```
# plot multiple lines with labels

num_samples=5
num_lines=5
for i in range(0,num_lines):
    x=np.linspace(2,10,num_samples)
    y=(i/num_lines)*x + 1
    _=plt.plot(x,y,label="slope_"+str(i))
_=plt.legend()
```

executed in 179ms, finished 18:09:25 2020-05-24



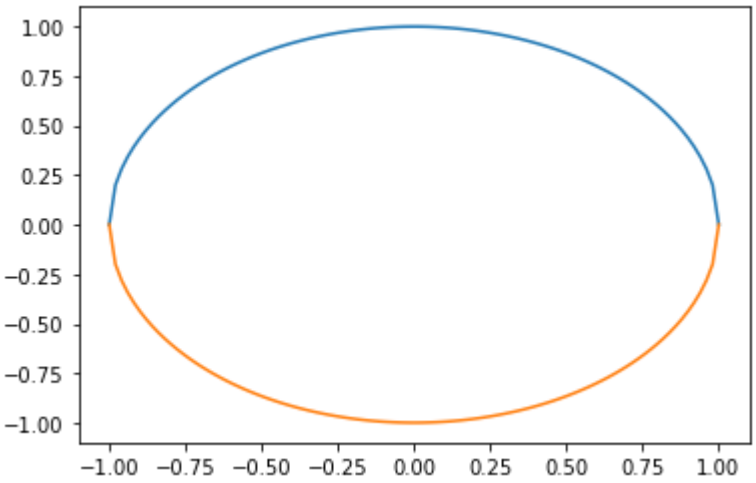
In [32]:

```
# plot a circle
num_samples=100
x=np.linspace(-1,1,num_samples)
y=np.sqrt(1-x**2)
plt.plot(x,y)
plt.plot(x,-y)
```

executed in 183ms, finished 18:09:25 2020-05-24

Out[32]:

```
[<matplotlib.lines.Line2D at 0x119614320>]
```

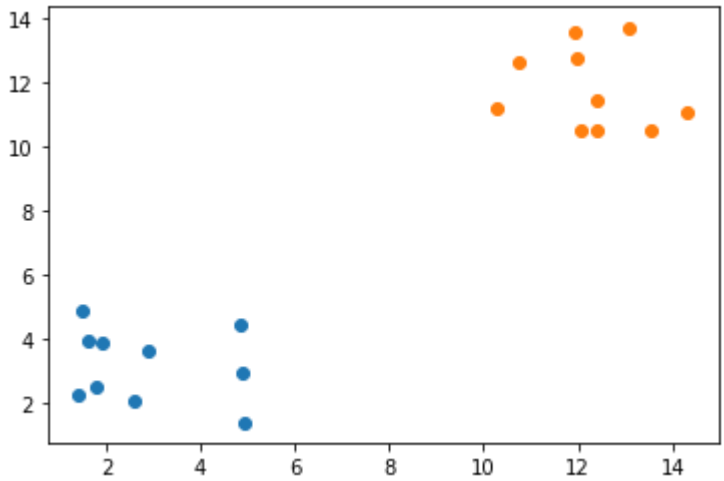


In [33]:

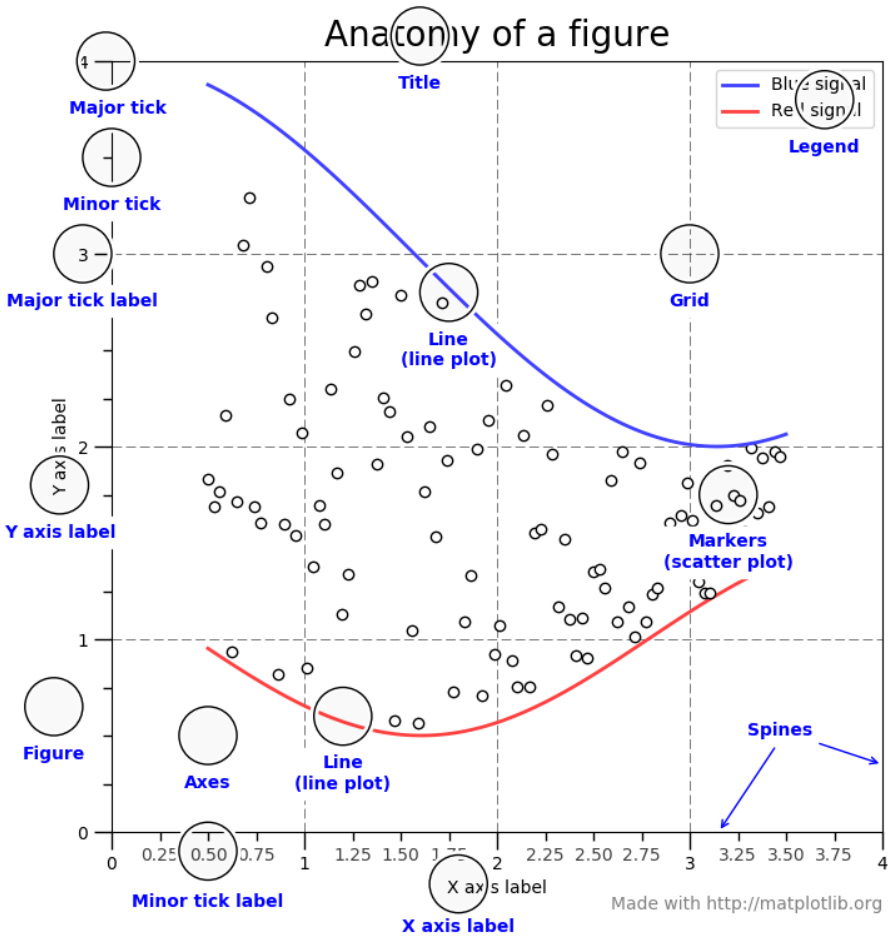
```
# scatter plot
num_samples=10
data=np.random.uniform(1,5,(num_samples,2))
print(data.shape)
plt.scatter(data[:,0],data[:,1])

num_samples=10
data=np.random.uniform(10,15,(num_samples,2))
print(data.shape)
_=plt.scatter(data[:,0],data[:,1])
```

executed in 138ms, finished 18:09:26 2020-05-24

```
(10, 2)
(10, 2)
```



2.0.2 Anatomy of a figure



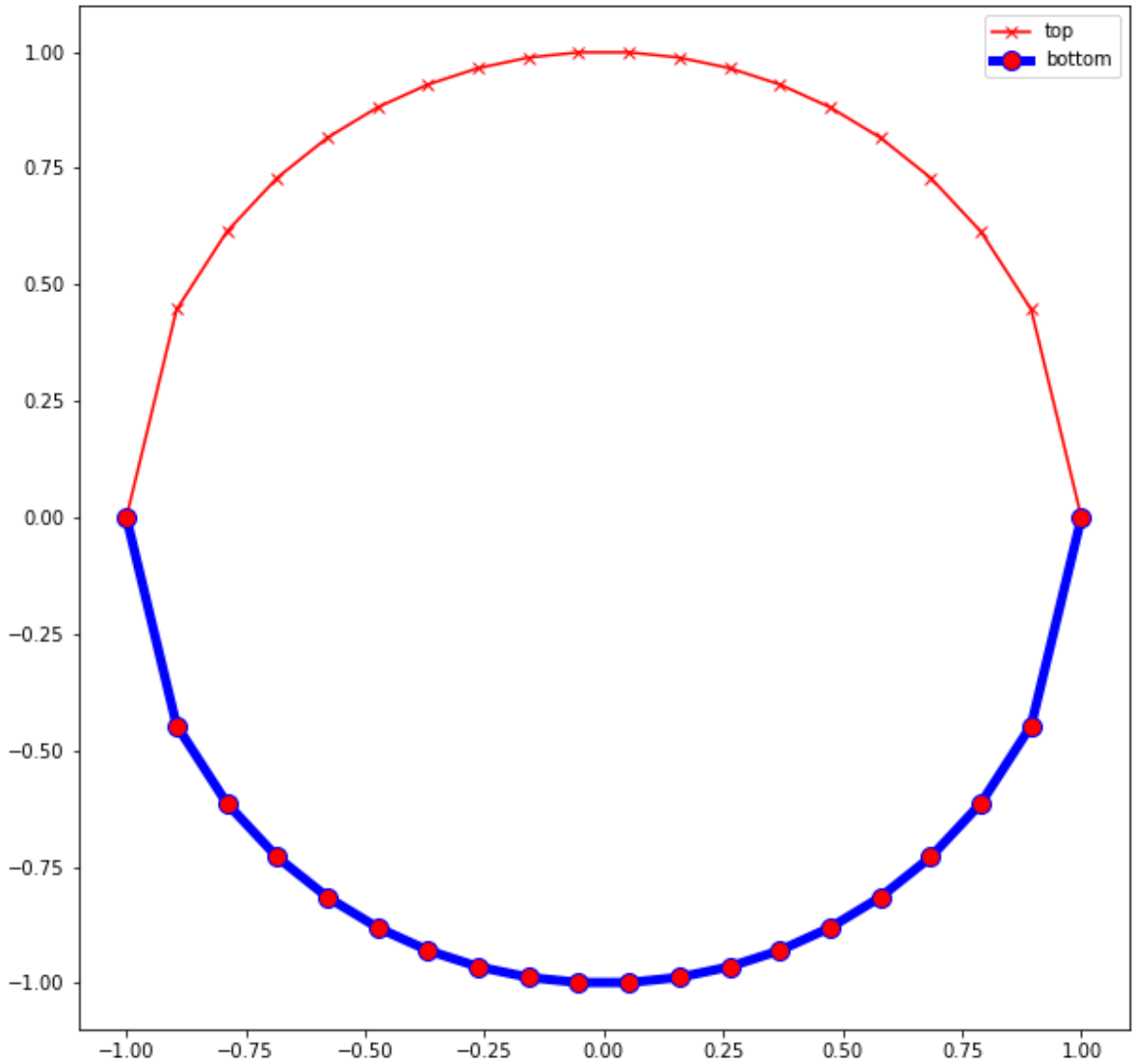
Source : https://matplotlib.org/tutorials/introductory/usage.html (https://matplotlib.org/tutorials/introductory/usage.html)

2.0.3 Changing the default formatting of your figure

In [34]:

```
# line labels, colors and markers
# adding a legend
# fmt
num_samples=20
x=np.linspace(-1,1,num_samples)
y=np.sqrt(1-x**2)
plt.figure(figsize=(10,10))
plt.plot(x,y,label="top",marker="x",color="r")
plt.plot(x,-y,label="bottom",marker="o",color="b",linewidth="5",markersize=10,mark
_=plt.legend()
```

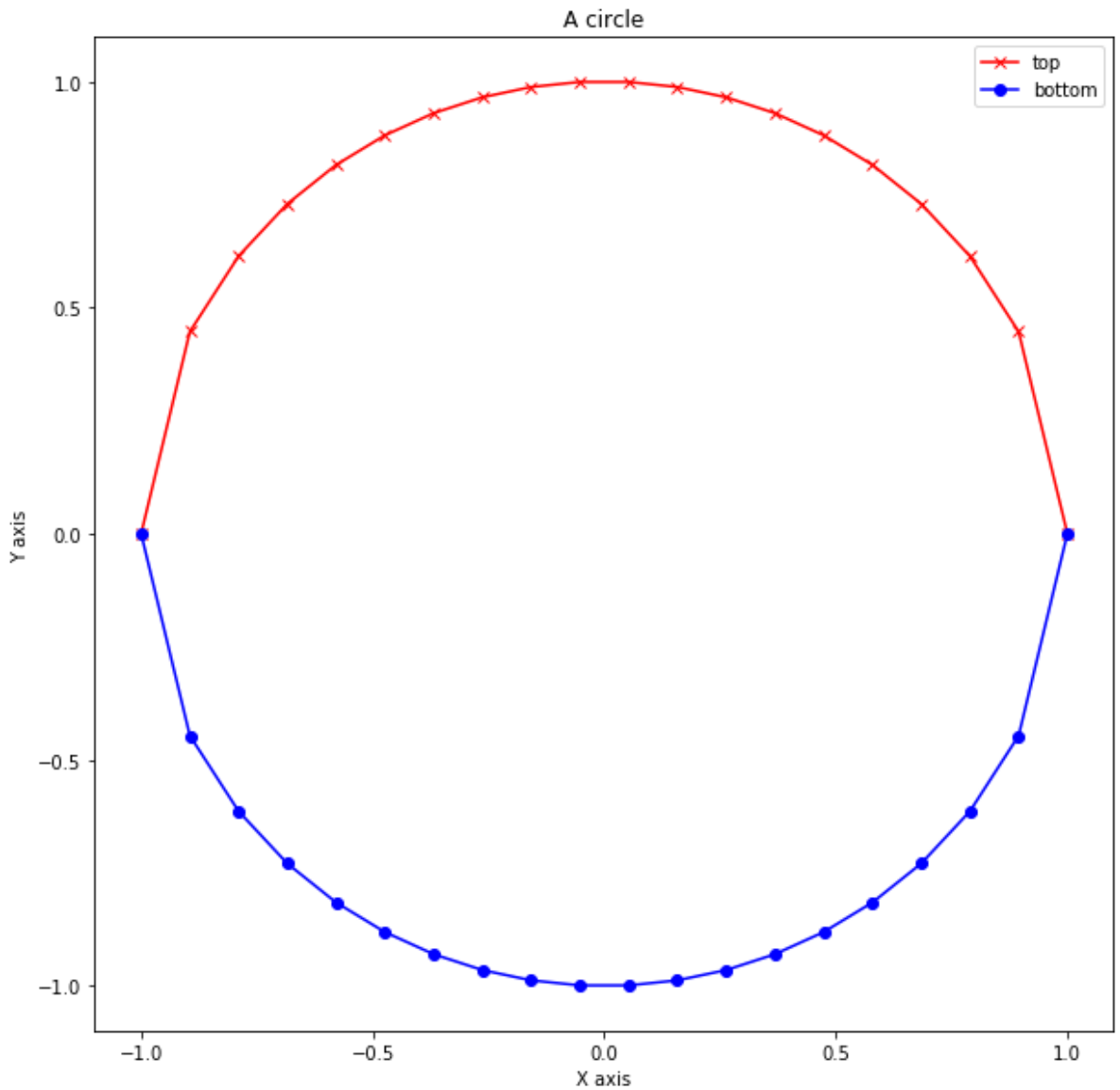
executed in 273ms, finished 18:09:26 2020-05-24



In [35]:

```
# title, ticks, labels, grid
num_samples=20
x=np.linspace(-1,1,num_samples)
y=np.sqrt(1-x**2)
plt.figure(figsize=(10,10))
plt.plot(x,y,label="top",marker="x",color="r")
plt.plot(x,-y,label="bottom",marker="o",color="b")

plt.title("A circle")
plt.xlabel("X axis")
plt.ylabel("Y axis")

plt.xticks(np.linspace(-1,1,5))
plt.yticks(np.linspace(-1,1,5))

_=plt.legend()
```

executed in 223ms, finished 18:09:26 2020-05-24

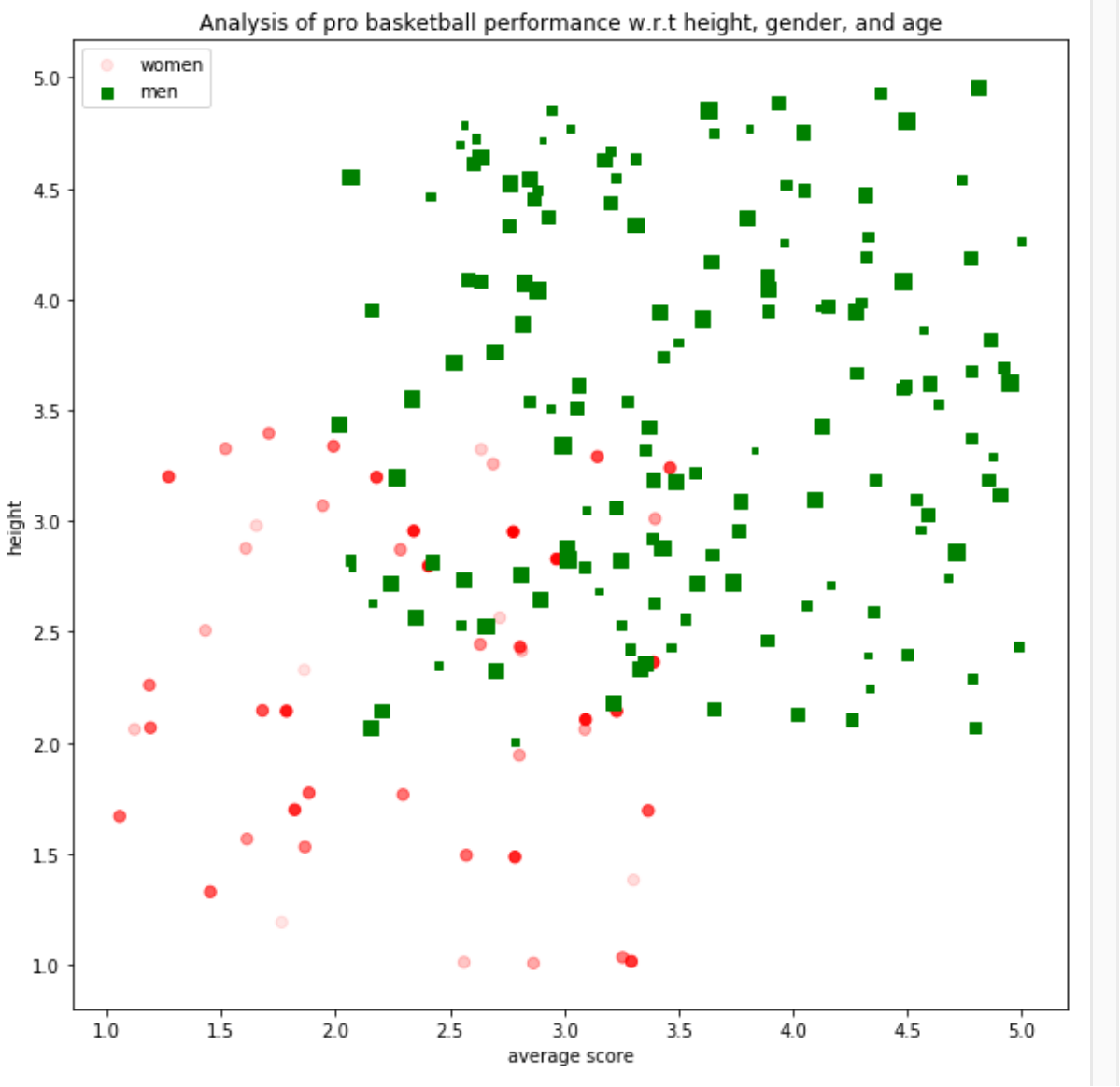


In [36]:

```
# scatter markers
# scatter size

# scatter plot
num_samples=50
data=np.random.uniform(1,3.5,(num_samples,2))
print(data.shape)

alphas = np.linspace(0.1, 1, num_samples)
rgba_colors = np.zeros((num_samples,4))
# for red the first column needs to be one
rgba_colors[:,0] = 1.0
# the fourth column needs to be your alphas
rgba_colors[:, 3] = alphas

plt.figure(figsize=(10,10))
plt.scatter(data[:,0],data[:,1],c=rgba_colors,s=40,marker="o",label="women")

num_samples=150
data=np.random.uniform(2,5,(num_samples,2))
print(data.shape)
markersizes=np.linspace(10,80,num_samples)
plt.scatter(data[:,0],data[:,1],color="g",s=markersizes,marker='s',label="men")

plt.title("Analysis of pro basketball performance w.r.t height, gender, and age")
plt.xlabel("average score")
plt.ylabel("height")
_=plt.legend()
```

executed in 318ms, finished 18:09:26 2020-05-24

```
(50, 2)
(150, 2)
```



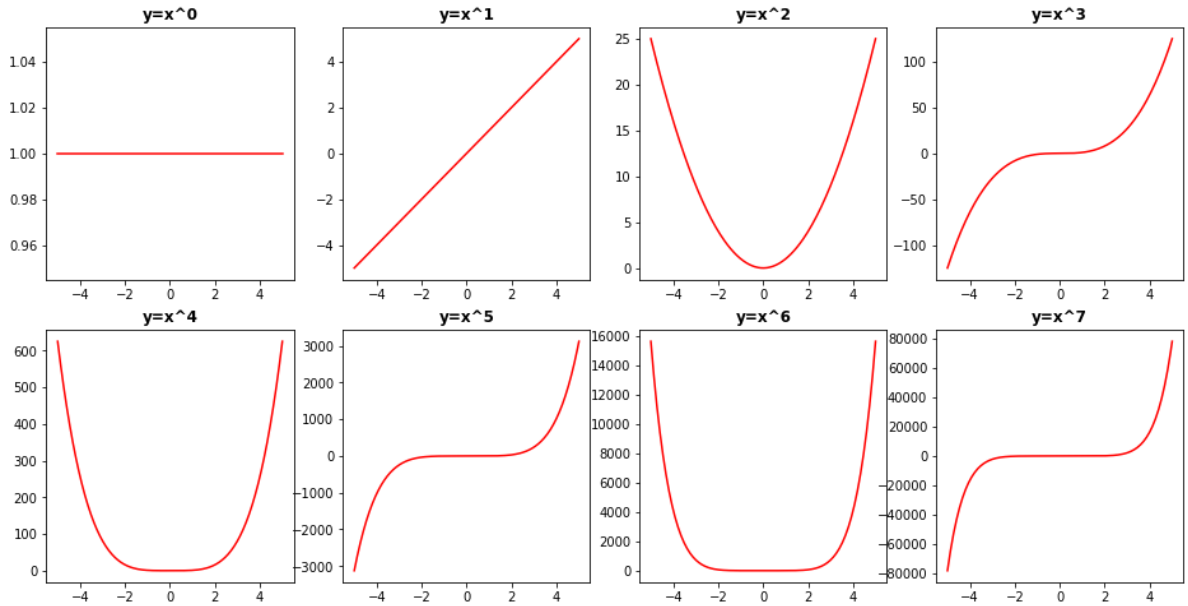
- These are just some examples, you can customize virtually anything in matplotlib - Google and Stackoverflow are your friends for this!

2.0.4 Subplots

In [37]:

```
import math

num_samples=150

# subplots example
NUM_PLOTS=8
num_columns=4
num_rows=NUM_PLOTS//4
fig,axes=plt.subplots(num_rows,num_columns,squeeze=False,figsize=(4*num_columns,4*
for i in range(0,NUM_PLOTS):
    myax=axes[math.floor(i/num_columns)][i%num_columns]
    myax.set_title("y=x^"+str(i),fontweight="bold",size="12")
    x=np.linspace(-5,5,num_samples)
    y=x**i
    myax.plot(x,y,color="r")
```

executed in 910ms, finished 18:09:27 2020-05-24



3 More numpy examples

3.0.1 Example 1 : Drawing a regular polygon

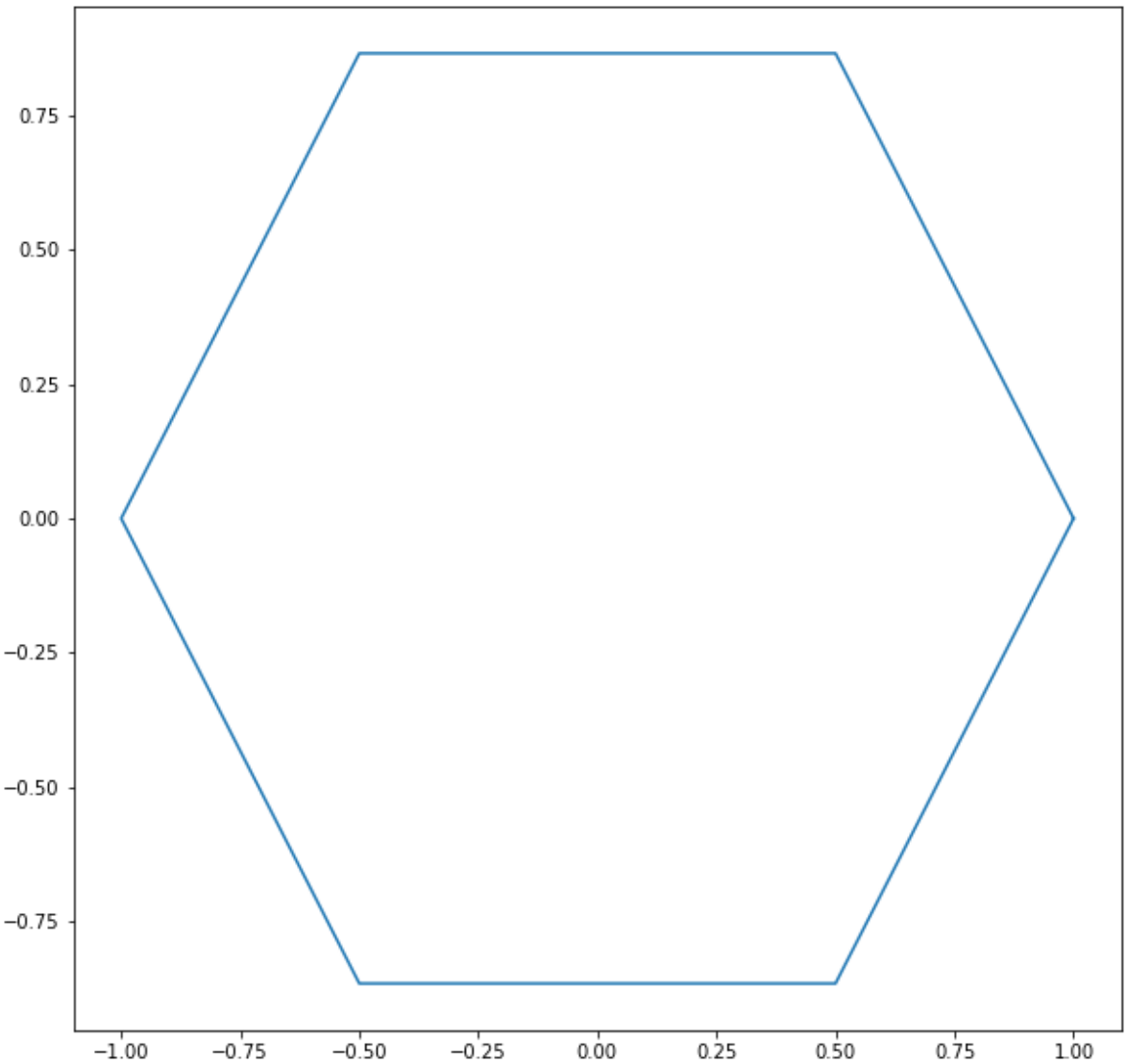
In [38]:

```
# generate points in order (theta) and plot
num_sides=6
radius=1
theta=2*np.pi/num_sides * np.arange(0,num_sides+1,1)
x=radius * np.cos(theta)
y=radius * np.sin(theta)
plt.figure(figsize=(10,10))
plt.plot(x,y)
```

executed in 253ms, finished 18:09:28 2020-05-24

Out[38]:

```
[<matplotlib.lines.Line2D at 0x1196358d0>]
```



3.0.2 Example 2 : Manipulating Images

In [39]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

executed in 2ms, finished 18:09:28 2020-05-24

In [40]:

```
# load image into numpy array
img=mpimg.imread("./car1.jpeg")
print(img,img.dtype,img.shape)
plt.figure(figsize=(10,10))
plt.imshow(img)
```

executed in 676ms, finished 18:09:28 2020-05-24

```
  [140 122 108]
  ...
  [ 93  64  56]
  [ 87  57  47]
  [ 93  61  50]]

 [[135 117 103]
  [139 121 107]
  [141 123 109]
  ...
  [ 88  60  49]
  [ 91  61  51]
  [ 91  59  48]]

 [[130 112  98]
  [133 115 101]
  [134 116 102]
  ...
  [ 89  59  49]
  [ 96  63  54]
```

In [41]:

```
# rgb to greyscale
# scale b/w 0 to 1
avg_image=np.sum(img,axis=2)/3
avg_image=avg_image.astype('uint8')
print(avg_image,avg_image.dtype,avg_image.shape)
plt.figure(figsize=(10,10))
plt.imshow(avg_image,cmap='gray', vmin=0, vmax=255)
```

executed in 554ms, finished 18:09:29 2020-05-24

```
[[116 121 123 ...  71  63  68]
 [118 122 124 ...  65  67  66]
 [113 116 117 ...  65  71  58]
 ...
 [ 48  46  43 ...  47  53  52]
 [ 42  42  50 ...  44  46  50]
 [ 38  39  48 ...  46  46  49]] uint8 (914, 1024)
```

Out[41]:

```
<matplotlib.image.AxesImage at 0x1042b8e48>
```

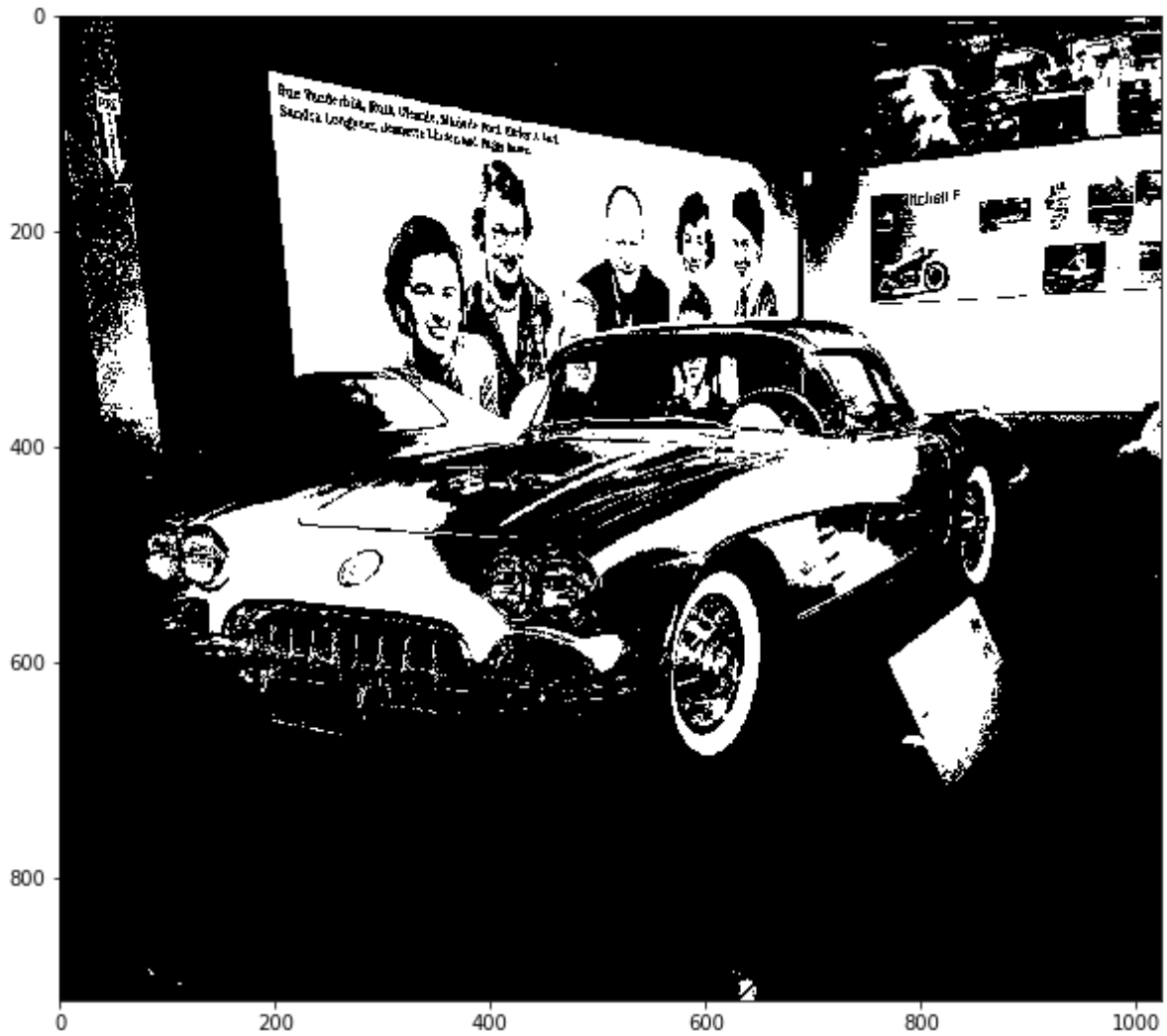
In [42]:

```
# discretize greyscale
num_levels=2
block_size=255/(num_levels-1)
disc_img=np.round((avg_image/block_size))
disc_img=np.round(disc_img*block_size)
disc_img=disc_img.astype('uint8')
plt.figure(figsize=(10,10))
plt.imshow(disc_img,cmap='gray', vmin=0, vmax=255)
```

executed in 306ms, finished 18:09:29 2020-05-24

Out[42]:

```
<matplotlib.image.AxesImage at 0x104316e80>
```



In [43]:

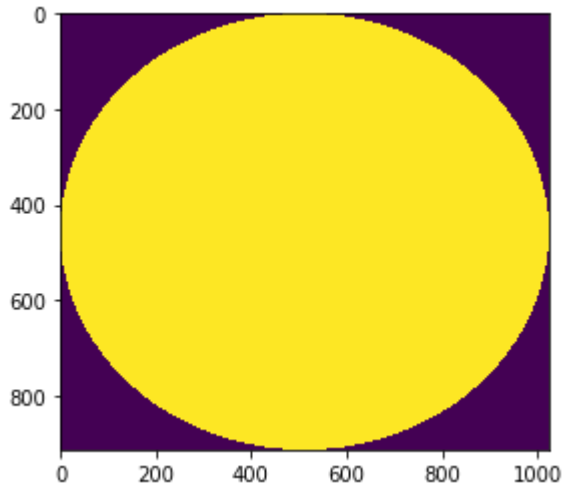
```
# frame

x=np.linspace(-1,1,img.shape[1])
print(x.shape)
X=np.tile(x,reps=(img.shape[0],1))
print(X.shape)
y=np.linspace(-1,1,img.shape[0]).reshape(-1,1)
print(y.shape)
Y=np.tile(y,reps=(1,img.shape[1]))
print(Y.shape)
circle=X**2 + Y**2 <1  # circle if img.shape[0]==img.shape[1] otherwise ellipse
plt.imshow(circle*255)

framed_image=np.where(circle, avg_image, 0)
plt.figure(figsize=(10,10))
plt.imshow(framed_image,cmap='gray', vmin=0, vmax=255)
```

executed in 649ms, finished 18:09:30 2020-05-24

```
(1024,)
(914, 1024)
(914, 1)
(914, 1024)
```

Out[43]:

```
<matplotlib.image.AxesImage at 0x11c90ba58>
```




0
200
400
600
800
0
200
400
600
800
1000

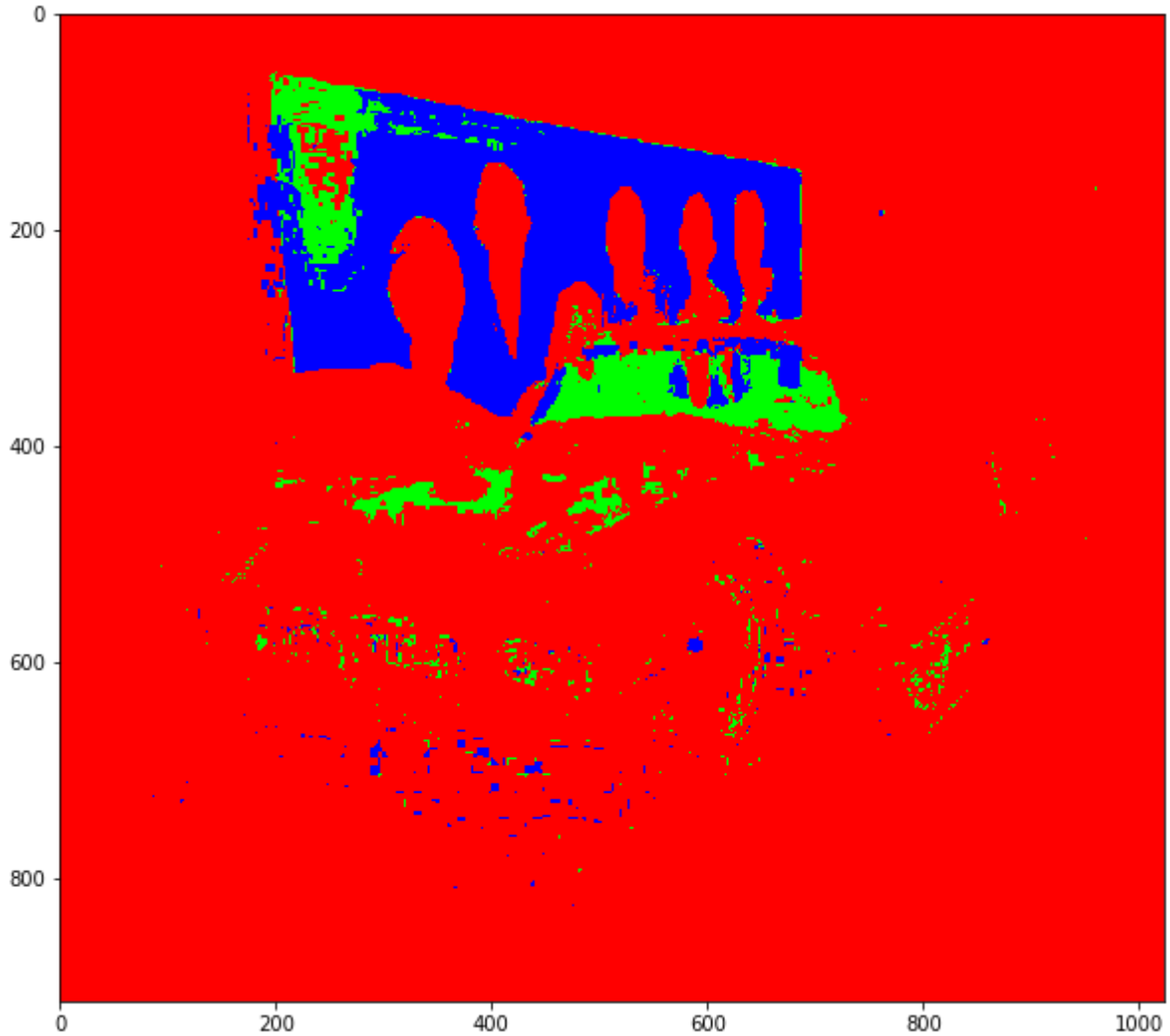
In [44]:

```
# discretize colors
red_channel=np.where(np.argmax(img,axis=2)==0,255,0)
red_channel=np.expand_dims(red_channel,axis=2)

green_channel=np.where(np.argmax(img,axis=2)==1,255,0)
green_channel=np.expand_dims(green_channel,axis=2)

blue_channel=np.where(np.argmax(img,axis=2)==2,255,0)
blue_channel=np.expand_dims(blue_channel,axis=2)

discrete_color=np.concatenate([red_channel,green_channel,blue_channel],axis=2)
plt.figure(figsize=(10,10))
plt.imshow(discrete_color)
```

executed in 399ms, finished 18:09:30 2020-05-24

Out[44]:

```
<matplotlib.image.AxesImage at 0x11989e550>
```



3.0.3 Example 3 : Writing Vectorized Code

- Matrix multiplication

In [45]:

```
d1=100
d2=100
d3=100
x=np.random.randint(1,5,d1*d2).reshape(d1,d2)
y=np.random.randint(1,5,d2*d3).reshape(d2,d3)
print(x)
print("\n\n")
print(y)
```

executed in 5ms, finished 18:09:30 2020-05-24

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



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

In [46]:

```
import time
```

executed in 2ms, finished 18:09:30 2020-05-24

In [47]:

```
# function for product of ith row and jth col
def calculate_entry(mat1,mat2,i,j):
    assert(mat1.shape[1]==mat2.shape[0])
    common_dim=mat1.shape[1]
    # ith row
    product_sum=0
    for posn in range(0,common_dim):
        product_sum+=mat1[i,posn]* mat2[posn,j]
    return product_sum
```

executed in 4ms, finished 18:09:30 2020-05-24

In [48]:

```
# 2 x loop + function
dim1 = x.shape[0]
dim2 = x.shape[1]
assert(dim2==y.shape[0])
dim3 = y.shape[1]
result1 = np.zeros((dim1,dim3))
start=time.time()
for i in range(0,dim1):
    for j in range(0,dim3):
        result1[i,j]=calculate_entry(x,y,i,j)
print(time.time()-start)
```

executed in 533ms, finished 18:09:31 2020-05-24

```
0.529094934463501
```

In [50]:

```
# replace function with slice + elementwise multiplication
dim1 = x.shape[0]
dim2 = x.shape[1]
assert(dim2==y.shape[0])
dim3 = y.shape[1]
result2 = np.zeros((dim1,dim3))
start=time.time()
for i in range(0,dim1):
    for j in range(0,dim3):
        result2[i,j]=np.sum(x[i,:]*y[:,j])
print(time.time()-start)
```

executed in 77ms, finished 18:09:45 2020-05-24

```
0.07281994819641113
```

In [51]:

```
# check same
np.all(result1==result2)
```

executed in 4ms, finished 18:09:46 2020-05-24

Out[51]:

```
True
```

In [52]:

```
# np.dot
start=time.time()
result3=np.dot(x,y)
print(time.time()-start)
```

executed in 4ms, finished 18:09:48 2020-05-24

```
0.0008111000061035156
```

In [53]:

```
# check same
np.all(result1==result3)
```

executed in 4ms, finished 18:09:49 2020-05-24

Out[53]:

```
True
```

- difference calculation

In [54]:

```
# function for product of ith row and jth col
def calculate_row_difference(mat1,mat2,i,j):
    assert(mat1.shape[1]==mat2.shape[1])
    feature_len=mat1.shape[1]
    # ith row
    diff_sum=0
    for posn in range(0,feature_len):
        diff_sum+=np.abs(mat1[i,posn]-mat2[j,posn])
    return diff_sum
```

executed in 4ms, finished 18:09:51 2020-05-24

In [55]:

```
# 2 x loop + function
num_rows1 = x.shape[0]
feature_len = x.shape[1]
num_rows3 = y.shape[0]
assert(feature_len==y.shape[1])
result2 = np.zeros((num_rows1,num_rows3))
start=time.time()
for i in range(0,num_rows1):
    for j in range(0,num_rows3):
        result1[i,j]=calculate_row_difference(x,y,i,j)
print(time.time()-start)
```

executed in 1.55s, finished 18:09:53 2020-05-24

```
1.5468480587005615
```

In [56]:

```
# difference vectorized
num_rows1 = x.shape[0]
feature_len = x.shape[1]
num_rows3 = y.shape[0]
assert(feature_len==y.shape[1])
result2 = np.zeros((num_rows1,num_rows3))
start=time.time()
for i in range(0,num_rows1):
    for j in range(0,num_rows3):
        result2[i,j]=np.sum(np.abs(x[i,:]-y[j,:]))
print(time.time()-start)
```

executed in 75ms, finished 18:09:53 2020-05-24

```
0.07019686698913574
```

In [57]:

```
np.all(result1==result2)
```

executed in 4ms, finished 18:09:53 2020-05-24

Out[57]:

```
True
```

In [58]:

```
# using broadcasting voodoo
num_rows1 = x.shape[0]
feature_len = x.shape[1]
num_rows2 = y.shape[0]
start=time.time()
mat1=x.reshape(num_rows1,1,feature_len)
mat2=y.reshape(1,num_rows2,feature_len)
diff=mat1-mat2
diff=np.abs(diff)
result3=np.sum(diff,axis=2)
print(time.time()-start)
```

executed in 17ms, finished 18:09:53 2020-05-24

```
0.012514829635620117
```

In [59]:

```
np.all(result1==result3)
```

executed in 6ms, finished 18:09:53 2020-05-24

Out[59]:

```
True
```