NYC DATA SCIENCE ACADEMY

Data Analysis in Python Numpy & Scipy

NYC Data Science Academy

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

NumPy: Overview

- NumPy is the fundamental package for scientific computing with Python.
  - Primitive data types are often collected and arranged in certain ways to facilitate analysis. A common example are vectors and matrices in linear algebra. Numpy provide easy tools to achieve it.
  - Numpy provides many functions and modules for scientific uses, such as linear algebra operations, random number generation, Fourier transform, etc.
- For full documentation, go to: http://docs.scipy.org/doc/.

NumPy: Overview

- As a module, you need to import it to make the functions it has accessible. You can do it by run the following code:

```
import numpy as np # later you can refer the package as np
```

- Whenever you need to call a function from Numpy, say `matrix()`, you could run:

```
np.matrix( arguments )     # The function matrix will be
                           # discussed
```

OVERVIEW

- NumPy
  - Ndarray
    - Subscripting and slicing
    - Operations
    - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

Data type: ndarray

- NumPy provides an N-dimensional array type, the *ndarray*. It can be described as a collection of elements of the same data type. We will discuss more about the creation of ndarray, but for the illustrating purpose, let's construct a simple example with the function `array()`

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

Data type: ndarray

- We see that `array()` takes a list as an argument. Then `my_ary` is a ndarray with three elements ordered exactly the same as the list inputted.

```
type(my_ary)
numpy.ndarray
my_ary
array([1,2,3])
```

- As you might suspect, each element in `my_ary` can be accessed by integer index. Details of indexing will be provided later.

```
print my_ary[0]  # In Python, first index is 0
print my_ary[1]
1
2
```

Data type: homogeneity and casting

- The type ndarray is flexible because it supports great variety of primitive data types (list of data types) subject to the homogeneity condition, which means every element in ndarray have the same type. The type of the elements can be accessed as an attribute of an ndarray.

```
my_ary.dtype
dtype('int64')
```

- When the elements are of string types, it print the dtype automatically. The letter 'S' stands for string and the number followed represents the largest number of letters among all the elements.

```
np.array(['a', 'ab'])
array(['a', 'ab'], dtype='|S2')
```

Data type: homogeneity and casting

- If the list inputted to the function `array()` is of mixed types, Numpy (upcast) cast all elements to a common (superclass) class:

```
np.array([1.0, 2, 3])
array([1.0, 2.0, 3.0])
# cast to float; The two integers included becomes
# float to meet the homogeneous condition.

np.array(['a', 2, 3])
array(['a', '2', '3'], dtype='|S1')
# Again for the homogeneity, all are casted to string.
```

Creating data array: 2 dimensional

- We have already seen a way to generate ndarray by plugging a list into the function `array()`. This method can be actually generalized to create multidimensional array by plugging nested list instead of simple list into the `array()` function.

```
simple_lst = [1,2]
nested_lst = [[1,2], [3,4]]
multi_ary = np.array(nested_lst)
array([[1, 2],
       [3, 4]])
```

- Recall that nested list is a list having lists as elements. In this particular case, multidimensional ndarray is designed to mimic mathematical matrices, so each inner list in `nested_lst` is like a row in a matrix.

Creating data array: higher dimension

- We can go one step further by plugging a list of multidimensional arrays into `array()`. This results in even higher dimension of an array. We will only do creating (here) and indexing for them. Most emphasis are put on the one and two dimensional arrays.

```
nested_lst2 = [[5,6], [7,8]]
multi_ary2 = np.array(nested_lst2)
higher_dim = np.array([multi_ary, multi_ary2])
higher_dim
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

Creating data array: Regular arrays creation

- So far we have been creating arrays mainly by specifying every element. When creating larger arrays this is not practical. Numpy provides functions which create functions by certain rules. We discuss some of them below.
- The simplest way to create a Numpy array is by the function `arange()`.

```
# np.arange(n) return an array from 0 to n-1
np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Plugging float into `arange()` result in array upto the greatest number smaller than the float.

```
np.arange(9.5)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Creating data array: Regular arrays creation

- We don't have to always start from 0. The first argument below indicates the array starts from 2 and the second argument indicates it stops at 9 (10-1).

```
np.arange(2, 10)
array([2, 3, 4, 5, 6, 7, 8, 9])
```

- And step between terms doesn't need to be 1. Below we specify the step to be 3 as the third argument, therefore 2 + 3 = 5, then 5 + 3 = 8. Since the next one 8 + 3 = 11, which is greater than the upper bound 9, so it stops at 8.

```
np.arange(2,10,3)
array([2, 5, 8])
```

Creating data array: Regular arrays creation

- A very similar method is `linspace()`. It takes the start (first argument), the end (second argument) and the desired length (last argument).

```
np.linspace(0,10, 51)
array([  0. ,   0.2,   0.4,   0.6,   0.8,   1. ,   1.2,
         1.4,   1.6,   1.8,   2. ,   2.2,   2.4,   2.6,
         2.8,   3. ,   3.2,   3.4,   3.6,   3.8,   4. ,
         4.2,   4.4,   4.6,   4.8,   5. ,   5.2,   5.4,
         5.6,   5.8,   6. ,   6.2,   6.4,   6.6,   6.8,
         7. ,   7.2,   7.4,   7.6,   7.8,   8. ,   8.2,
         8.4,   8.6,   8.8,   9. ,   9.2,   9.4,   9.6,
         9.8,  10. ])
```

- Without the last argument, the length defaults 50.

Exercise 1

- Which of the following two arrays has all integer (this refer to the value, not data type) entries?
  - `np.linspace(0, 9, 3)`
  - `np.linspace(0, 9, 4)`
- As `array()`, we can change the datatype by specifying `dtype`. Run the following code, what do you find?
  - `np.linspace(0, 9, 4, dtype ='Int64')`
  - `np.linspace(0, 9, 3, dtype ='Int64')`

Creating data array: Regular arrays creation

- We might want to construct a constant array. The function ones() returns an array with as many 1 as the number plugged in.

```
np.ones(3)
array([1.0, 1.0, 1.0])
```

- We can construct all the constant array by multiply the desired constant to a array produced as above. The details of array operation will be discussed, but multiplying a number to an array results in every entry in the array multiplied by the constant.

```
3.1415 *np.ones(5)
array([ 3.1415,  3.1415,  3.1415,  3.1415,  3.1415])
```

Creating data array: Regular arrays creation

- We have seen from above that we can easily generate a zero array by times 0 to a array generated by the function `ones()`. However, Numpy also provides a function `zeros()`, which works in the same way as the function `ones()`:

```
np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

Index

- We often need to access a particular element (subscripting) in our array. We have already seen that ndarray is treated as an ordered sequence, whose entries are indexed by integers.

```
x = np.arange(1, 11)
```

- We may print out an arbitrary entry. For example, the 3rd element in x:

```
x[2]  # Python index from 0
3
```

- We may plug it into a function:

```
3**(x[2])
27
```

Index

- We may update an element with indexes:

```
x[2] = 100
x
array([  1,   2, 100,   4,   5,   6,   7,   8,   9,  10])
```

- Negative indexes select the element by the opposite order.

```
print x[-1]; print x[-2]; print x[-10]
10.0
9.0
1.0
# Notice that it does not start with "-0", OF COURSE!!
```

Exercise 2

- Initialize you x again with:

```
x = np.arange(1, 11)
```

- Run `x[2]=3.0`. Which entry of the array x will be updated?
- Run `x[2]=3.1`. Which entry of the array x will be updated?
- Run `x.astype(float)` to change the type to float, then update the third element to 3.1

Index

- We often need to subscript in a higher dimensional array. We start by constructing one:

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

- As we mentioned before, 2-dimensional ndarray can be thought as a nested list. Therefore, to access the number 2, it is the element 1 in the list 0:

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

Index

- In this way, we may slice a particular row by selecting an inner list:

```
high_x[1]
array([4, 5, 6])
```

- A 2-dimensional array can be also taken for a matrix.

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

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Index

- Therefore as matrices, a 2-dimensional array can be indexed a pair of integers -- row indexes and columns indexes. Conventionally, the first index is for row, the second for column.

```
print high_x[0,0]; print high_x[0,1]; print high_x[0,2]
print high_x[1,0]; print high_x[1,1]; print high_x[1,2]
1
2
3
4
5
6
```

Index

- We may then slice part of the second row by specifying the row index and the range of the columns.

```
high_x[1,0:2]
array([4, 5])
```

- When specifying index to be from the first entry, we can leave it blank.

```
high_x[1,:2]
array([4, 5])
```

Shape

- Again as a matrix, the shape of an array can be denoted by the number of rows and columns it has. And again, first rows and then columns. For example, consider the following is a 2 by 3 matrix.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

- For an array, the shape (or dimension in more matrix convention) can be seen by calling the `.shape` attribute.

```
high_x.shape
(2, 3)
```

Shape

- Now we understand the notation of the shape. We may discuss the regular creation of 2-dimensional arrays. We start with generating a constant matrix by the functions `ones()`.

```
np.ones([2,3])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

- The function `zeros()` works in the same way:

```
np.zeros([3,2])
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Shape

- We may also rearrange the shape of a 1-dimensional array to obtain a higher dimensional one. This can be done by a method `reshape()`

```
x = np.arange(8)
x.reshape([2,4])
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

- This can also be achieved by assigning the desired value to the attribute `.shape`

```
x = np.arange(8)
x.shape = (2,4)
```

Shape

- The method `reshape()` actually allows two different ordering. The one we see from the previous page fill up row by row. This is specified by the default value of the keyword parameter `order='C'`:

```
x = np.arange(8)
x.reshape([2,4], order='C')
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

- We may instead use `order='F'`, so the matrix be filled up column by column.

```
x = np.arange(8)
x.reshape([2,4], order='F')
array([[0, 2, 4, 6],
       [1, 3, 5, 7]])
```

Shape

- All the creation method we talked about can be of course generalized to even higher dimension, here is an example

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

       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])
```

- Audience are invited to try on the other creation method we mentioned for higher dimensional cases.

Exercise 3

- Create a 4 by 5 five matrix with all the entries 8.
- Create an array corresponding to the matrix below:

$$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \end{pmatrix}$$

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

Arithmetic operations

- We often want to do math on an array. Addition and scalar multiplication are widely used. They work as expected:

```
x = np.array([1,2])
y = np.array([3,4])
z = np.array([5,6])
x+y+z
array([ 9, 12])

5*x
array([5, 10])
```

Arithmetic operations

- Addition is actually an example of pointwise operations. Here is another example:

```
y*z
array([ 15, 24])
```

- All the arithmetic operators can work pointwisely.

```
x**y
array([1, 16])
y/x   # (Integer) Division
array([3, 2])
y-x
array([2, 2])
z%y
array([2, 2])
```

Arithmetic operations

- On the other hand, scalar product is an special case of broadcasting.

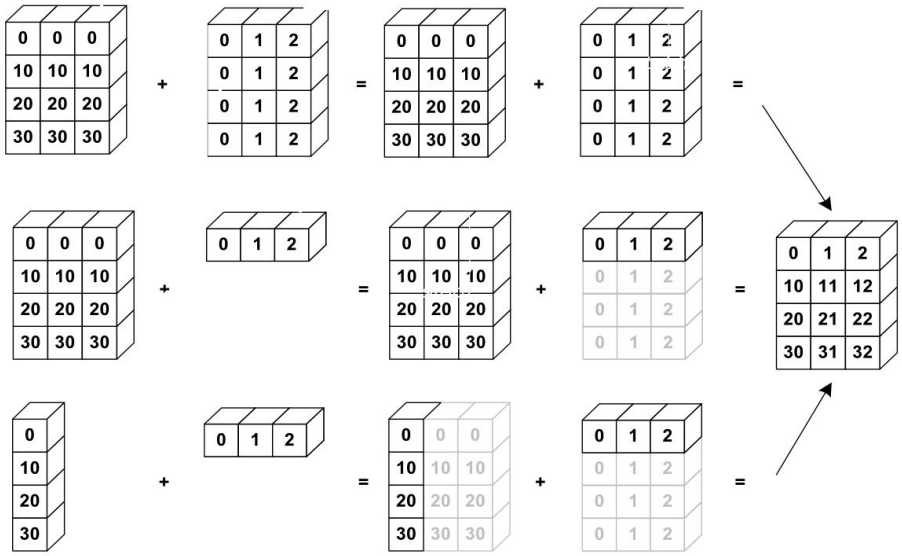
```
x+3         # Addition can be broadcasted
array([4, 5])
# Add 3 to each position of x
```

- All the arithmetic operators can be broadcasted similarly

```
x**2
array([1, 4])
x/2   # (Integer) Division
array([0, 1])
y-3
array([0, 1])
z%3
array([2, 0])
```

Broadcast operations

- More general broadcasting rule can be applied when the arrays have different shape. We leave the details to the audiences.



- The general broadcasting rules can be found via:
  - http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Comparison operations

- Comparison operators generates Boolean arrays. As arithmetic operators, they also follow the broadcasting rule.

```
x>1
array([False,  True], dtype=bool)
```

- All the comparison operators can be broadcasted similarly.

```
x < 1
array([False, False], dtype=bool)
y<=3
array([ True, False], dtype=bool)
z>=6
array([False,  True], dtype=bool)
z==5
array([ True, False], dtype=bool)
```

Comparison operations

- Comparison operators can be applied pointwisely to multiple arrays.

```
x>y
array([False,  False], dtype=bool)
```

- All the operators operators can be broadcasted similarly.

```
x < y
array([True, True], dtype=bool)
y<=z
array([True, True], dtype=bool)
z>=x
array([True,  True], dtype=bool)
z==x
array([False, False], dtype=bool)
```

Combining and Aggregating Boolean Arrays

- In many cases we want to combine arrays with and or or componentwise.

```
b_1 = np.array([True, True, True])
b_2 = np.array([True, True, False])
b_3 = np.array([True, False, False])
b_4 = np.array([False, False, False])

print np.logical_or(b_2, b_3)
print np.logical_and(b_2, b_3)
[ True  True False]
[ True False False]
```

Combining and Aggregating Boolean Arrays

- We also very often wants to aggregate a boolean array:

```
print b_1.all()
print b_2.all()
True
False

print b_3.any()
print b_4.any()
True
False
```

Fancy indexing

- We often need to filter an array according to some condition. This can be done by two steps.
  - Generate Boolean array

```
x==1
array([ True, False], dtype=bool)
```

  - Slice the array with the Boolean array

```
x[np.array([ True, False])]
array([1])
```

  - The two steps can be actually combined

```
x[x==1]
array([1])
```

Fancy indexing

- Fancy indexing can be applied to (higher) 2 dimensional arrays. However, slicing with fancy index drop the structure to the 1 dimensional array.

```
high_x = np.array([[1,2,3],[4,5,6]])
high_x[high_x>=3]
array([3, 4, 5, 6])
```

- However, assignment with fancy indexing doesn't change the shape.

```
high_x[high_x>=3]=10
high_x
array([[ 1,  2, 10],
       [10, 10, 10]])
```

Exercise 4

- Run the code below to create arrays.

```
ary_1= np.ones([3,2])
ary_2=np.arange(1,7).reshape(3,2)
```

  - Sum up the arrays.
  - Add 1 to the first column of ary_1, and add 2 to the second column of ary_1.
  - Update ary_2: change any number greater than 4 to 2.5.

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

Matrix type

- We mentioned that the 2 dimensional arrays can be indexed with matrix convention. Numpy actually provides a function `matrix()` that changes the type ndarray to *numpy.matrixlib.defmatrix.matrix* (we call them *matrix* for simplicity hereafter).

```
x = np.matrix([1,2])
type(x)
numpy.matrixlib.defmatrix.matrix
```

Exercise 5

- We illustrate an important side issue here. First create an array by `ary=np.array([1,2,3,4])`
- Create two matrices, `m1=np.matrix([1,2,3,4])` and `m2=np.mat([1,2,3,4])`. These are two different functions generate matrices, but when you print out m1 and m2, do they look different?
- Create the two matrix in different ways, `m1=np.matrix(ary)` and `m2=np.mat(ary)`. Print them out again, do you see any difference?
- Now, execute `ary[1]=0`. Now print out m1 and m2 again, are they still the same?
- Remark: This illustrate a common issue when making a copy (m2 is a transformation of a copy of ary). We often thought we create a new object by copying but in fact they are often just two different names of one object. Changing one of them changes the other.

Matrix type

- Once an array is converted to a matrix, it is taken for a 2-dimensional object. For example, we can access the second entry of an array by:

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

Matrix type

- However, if we change y into a matrix, it no longer works:

```
y = np.matrix(y)
y[1]
---------------------------------------------------------------
IndexError                Traceback (most recent call last)
<ipython-input-145-50d607d5521c> in <module>()
      1 y = np.matrix(y)
----> 2 y[1]
/Users/LukeLin/anaconda/lib/python2.7/site-
packages/numpy/matrixlib/defmatrix.pyc in __getitem__(self,
index)
    316
    317         try:
--> 318             out = N.ndarray.__getitem__(self, index)
    319         finally:
    320             self._getitem = False

IndexError: index 1 is out of bounds for axis 0 with size 1
```

Matrix type

- Since y is now taken for 2-dimensional (Notice the extra bracket)

```
y
matrix([[3, 4]])
```

- The expression `y[1]` now indicates the second row of y, which it does not have. To access the element 4, we need to select the second element in the first row by:

```
y[0, 1]
4
```

- Multiple inner lists are considered multiple rows:

```
z = np.matrix([[3],[4]])
```

Matrix multiplication

- Another important difference between arrays and matrices is the multiplication. While the comparison and the other arithmetic operations still works pointwisely, multiplication (and therefore raising power) does not. Run the code below and look at the error message.

```
x*y
```

- The type matrix is for linear algebra, in which the matrix multiplication is always "a row times a column":

$$\begin{pmatrix}1 & 2 & 3\end{pmatrix} \times \begin{pmatrix}1\\2\\3\end{pmatrix} = \begin{pmatrix}1+4+9\end{pmatrix} = \begin{pmatrix}14\end{pmatrix}$$

Exercise

- Try the matrix multiplication. What happen to the ones highlighted?

$$\begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \times \begin{pmatrix} 2 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \times \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

Matrix multiplication

- Both our x and y transformed from ndarray into matrix type, which is taken for a row vector (matrix with 1 row, if you want). To change the later to a column vector (matrix with 1 column), we may use the method `.T`

```
y
matrix([[3, 4]])
y.T
matrix([[3],
        [4]])
```

- Then we may do the multiplication

```
x*y.T
matrix([[11]])
```

Identity

- An important special matrix is called **identity**. An identity is a square matrix (the number of the rows and the columns are the same), all of whose entries are 0, except the diagonal entries are 1. A identity array can be created by specifying the dimension in the function `eye()`:

```
np.eye(3)
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

- We then change it into a matrix type

```
Id = np.matrix(np.eye(3))
```

Identity

- The importance of the identity matrix lies on the fact any matrix multiply by the identity matrix (with the dimension so that a multiplication is applicable) remains unchanged.

```
my_mat = np.matrix(range(9))
my_mat = my_mat.reshape([3,3])
my_mat
matrix([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
Id*my_mat
matrix([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.]])
```

Identity

- Actually, the identity matrix we created above can be applied to more than just 3 by 3 matrices. Whenever the dimension match:

```
my_vec1 = (np.matrix(range(3))).reshape([3,1])
Id*my_vec1
matrix([[ 0.],
        [ 1.],
        [ 2.]])

my_vec2 = (np.matrix(range(3))).reshape([1,3])
my_vec2*Id
matrix([[ 0.,  1.,  2.]])
```

Inverse

- Identity matrix is analogous to 1 in real number, any number times 1 remains unchanged. A nonzero real number has a reciprocal, the analogy for a matrix is called an *inverse*. For example,

```
m1 = np.matrix([[1,2],[0,1]])
m1
matrix([[1, 2],
        [0, 1]])

m2 = np.matrix([[1,-2],[0,1]])
m2
matrix([[ 1, -2],
        [ 0,  1]])
```

Inverse

- If we compute the product of m1 and m2 (in both orders), we get:

```
m1*m2
matrix([[1, 0],
        [0, 1]])

m2*m1
matrix([[1, 0],
        [0, 1]])
```

- We remark that if a matrix have an inverse, it has *the* unique inverse. Therefore, for the example above one can say the inverse of m1 (m2) is m2 (m1).

Inverse

- For a Numpy matrix (with an inverse), there is a method returning the inverse matrix:

```
m1.I  # See below, it's actually m2
matrix([[ 1., -2.],
        [ 0.,  1.]])

m2.I  # See below, it's actually m1
matrix([[ 1.,  2.],
        [ 0.,  1.]])
```

Exercise 6

- How do you find a tuple of x, y and z satisfying all the equations below?

$$\begin{aligned} 3x + 2y - \ \ z &= \ \ 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - \ \ z &= \ \ 0 \end{aligned}$$

- Observe that the equation can be written as:

$$\begin{pmatrix} 3 & 2 & -1 \\ 2 & -2 & 4 \\ -1 & 1/2 & -1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 0 \end{pmatrix}$$

From here, can we use inverse to solve for x, y and z?

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

SciPy: Overview

- SciPy is a package for scientific computing and technical computing with Python.
  - SciPy builds on the NumPy array object and is part of the NumPy stack which includes tools like Matplotlib and pandas.
  - SciPy contains modules for stats (which we will focus on), integration, optimize, linear algebra, interpolation, special functions, FFT, signal, image processing, ODE solvers and more.
- For full documentation, go to: http://docs.scipy.org/doc/.

Sample data set

- Before we begin the class in SciPy, we import the modules we need for demonstration. We import datasets from “sklearn” module simply because we need the sample data set -- iris.

```
import numpy as np
from scipy import stats

from sklearn import datasets
iris_raw = datasets.load_iris()
```

SciPy: the stats module

- The stats module contains variety of tools for statistical task, we will demonstrate the topic below:
  - Statistical functions.
  - Hypothesis test.
  - Random Sampling.
- You are invited to run the code below to see a list of functions stats provides.

```
import scipy
scipy.info(stats)
```

Sample data set

- The package sklearn (its real name is *scikit-learn*) is for machine learning. it contains a popular sample data set, iris, to demonstrate classification problem, which takes some known feature into account then predict the class each instance belongs to. In the iris data set, the known features are the length and the width of the petals, and the length and the width of the sepals. With those information, functions in sklearn can be used to predict the species of each instance. We will not discuss machine learning today, but we will use the sepal length and the pedal length as our example for numeric features and the species for categorical features.

```
iris_raw.keys()
```

Sample data set

- The features of iris data is recorded in `iris_raw.data` as a 2-dimensional array. There are 150 rows corresponding to 150 instances (flowers), 4 columns corresponds to 4 different features.

```
iris = iris_raw.data
iris[0:8,:]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2],
       [ 5.4,  3.9,  1.7,  0.4],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5. ,  3.4,  1.5,  0.2]])
```

Sample data set

- We will use the following three arrays

```
sepal_len = iris[:,0]
sepal_wid = iris[:,1]
petal_len = iris[:,2]
```

Sample data set

- The labels of iris data is recorded in `iris_raw.target` as an array.We manually picked some entries to illustrate:

```
iris_raw.target[[0,1,50,51,100,101]]
array([0, 0, 1, 1, 2, 2])
```

- We call the target “species”:

```
species=iris_raw.target
```

- Though the values are integer, they are actually representing the species, which can be seen from

```
iris_raw.target_names
array(['setosa', 'versicolor', 'virginica'],
      dtype='|S10')
```

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
      - Hypothesis test
  - Random Sampling
  - Distribution

SciPy: statistical functions

- We often want a quick descriptive stats when having new data. stats provides function to deal with both categorical and numerical features. The function `itemfreq()` returns the frequency of each class you have in a categorical feature.

```
stats.itemfreq(species)
array([[ 0, 50],
       [ 1, 50],
       [ 2, 50]])
```

SciPy: statistical functions

- Numerical features can be summarized by the function `describe()`:

```
stats.describe(sepal_len)
DescribeResult(nobs=150, minmax=(4.30, 7.90),
               mean=5.843, variance=0.68,
               skewness=0.31, kurtosis=-0.57)
```

- The function returns an special object scipy.stats.stats.DescribeResult. To access each of item, we can use either use the item name of integer index

```
stats.describe(sepal_len)[0]
150
stats.describe(sepal_len).minmax
(4.30, 7.90)
```

SciPy: statistical functions

- More on statistical functions can be found in the link.

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

SciPy: hypothesis test

- We often need to gain insight to the various behaviors existent in the population. In order to do that we need to form a testable hypothesis and derive the p-value (probability) that the hypothesis is valid. Usually this hypothesis we test on are called *null* hypothesis, and is tested for possible rejection under the assumption that it is true. The contrary to the null hypothesis is called the alternative hypothesis that would be retained if the null hypothesis is rejected.
- The stats module provides functions to perform a variety of hypothesis tests. We will demonstrate with some of most common ones.

One-sample t-test

- We use the one-sample t-test to investigate the difference between a sample and a proposed population mean. For example, if we want to know how likely that the population mean of the array petal_len is 10, we may use `ttest_1samp()`:

```
stats.ttest_1samp(petal_len, 10)
Ttest_1sampResult(statistic=-43.323240335498866,
                  pvalue=2.4562011895365267e-86)
```

- In this case, the p-value is extremely small, which indicates that the population mean from which the petal_len sample was drawn is UNLIKELY to be 10. Thus, it is “safe” to say that the mean of the population is not 10.

Two-sample t-test

- The two-sample t-test is used to test hypotheses on two samples having the same population mean. For example, if we want to know how likely that the population mean of the array sepal_len and the array petal_len are the same, we may use `ttest_ind()`:

```
stats.ttest_ind(sepal_len, petal_len)
Ttest_indResult(statistic=13.099504494510061,
                pvalue=2.8297338637366177e-31)
```

- In this case, the p-value is extremely small, which indicates that the population mean of sepal_len and that of petal_len are UNLIKELY to be the same.

ANOVA

- The two-sample t-test allows us to check whether the population mean is significantly different between two samples. What if we have more than two samples? Analysis of Variance is a way to treat this kind of cases.

```
stats.f_oneway(sepal_len, sepal_wid, petal_len)
F_onewayResult(statistic=237.45598625019167,
               pvalue=5.440492586968288e-71)
```

- The function f_oneway is used to implement one-way analysis of variance. In this particular case, since the p value is very small, it is UNLIKELY that all the samples have the same population mean.

SciPy: hypothesis tests

- More on hypothesis tests can be found from the page statistical functions, which we have seen before.

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - **Random Sampling**
  - Distribution

Random sampling

- We often want to generate array randomly for simulation or testing purpose. Both Numpy and SciPy provide functions for random sampling.
- List of Functions for Random Sampling in Numpy:
  - `randn(d0, d1, ..., dn)`: Each entry is from the "standard normal" distribution.
  - `rand(d0, d1, ..., dn)`: Random array with a given shape where each element is from Uniform [0, 1].
  - `randint(low, high, size)`: Random integers ranging from low (inclusive) to high (exclusive).

Random sampling

- List of Functions for Random Sampling in Numpy:
  - `random_integers(low, high, size)`: Random integers between low and high, both inclusive.
  - `random_sample(size)`: Random floats in the half-open interval [0.0, 1.0).
  - `choice(a[, size, replace, p])`: Generates a random sample from a given 1-D array. By default, `replace=True`.

Random Sampling: Birthday Problems

- If we randomly choose 20 people, what is the probability that some of them share the same birth day? For people who are familiar with probability, you might want to compute the probability that nobody sharing the same birthday and subtract that from 1. You are right that this is doable.
- We will approach a solution with another interesting method -- simulation. We know that there would be 366 different possible birthday, including Feb. 29. Assuming the probability for each date to be picked is the same (obviously this is not valid, we assume it for simplicity)

Random Sampling: Birthday Problems

- To simulate the situation, we may use the `choice()` function from the `numpy.random` submodule.

```
np.random.seed(1)
np.random.choice(range(366), size=20, replace=True)
array([ 37, 235,  72, 255, 203, 133, 335, 144, 129,
        71, 237, 281, 178, 276, 254, 357, 252, 156,
        50,  68])

np.random.choice(range(366), size=20, replace=True)
array([215, 241, 352,  86, 141,   7, 319, 317,  22,
       313,   1, 316, 209, 264, 216, 141, 115, 121,
        30,  71])
```

Random Sampling: Birthday Problems

- The first simulation has no duplicated dates and the second does have 141 twice. If we only simulate twice, we might think that the desired probability is 0.5!
- However, this is obviously very biased! What we want to do is to simulate many times and calculate the probability according to that. Below are the code for simulation.

Random Sampling: Birthday Problems

```
num_people=20
num_simu=int(1e3)
Bool = np.zeros(num_simu)
for i in range(num_simu):
    test = np.random.choice(range(366),\
                        size=num_people, replace=True)
    Bool[i] = (len(set(test))!=num_people)
np.mean(Bool)
0.405
```

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
  - Matrix and linear algebra
- SciPy
  - Stats Module
    - Statistical function
    - Hypothesis test
  - Random Sampling
  - Distribution

Distributions

- In the birthday problem, we made an (invalid) assumption that every possible outcome (birth date of each person) is identical. This made our simulation much easier.
- If this is not the case, we need to pay attention to the likelihood (I didn't use probability for some reason) of the occurrence of each outcome. This piece of information are provided by a mathematical object: **distribution**.

Binomial Distributions

- Binomial distribution is a very simple case where the probability of each possible outcome is not constant.
- A typical example of binomial distribution is: flipping a fair coin 5 times, what is the probability of getting certain amount of heads?
- The events we are considering is like below:

```
#### We will talk about this code a bit later

my_binom = stats.binom(5, 0.5)
np.random.seed(1)
my_binom.rvs(10)
array([2, 3, 0, 2, 1, 1, 1, 2, 2, 3])
```

Binomial Distributions

- The sequence above "simulates" the result of 10 experiments. In each experiment we toss a fair coin 10 times, and:
  - we get 2 heads in the first experiment.
  - we get 3 heads in the second experiment.
  - we get no head in the third experiment.
  - ...
- Obviously, the probability of getting 1 head and getting no head are different; and the possible outcomes are no head, 1 head, 2 heads, 3 heads, 4 heads and 5 heads.

Binomial Distributions

- Let's do a little math:
  - The probability of having no head out of 5 tossing is

$$(0.5)^5 = 0.03125.$$

  - The probability of having one head out of 5 tossing is

$$C_1^5 \cdot (0.5)^5 = 0.15625$$

  - The probability of having two heads out of 5 tossing is

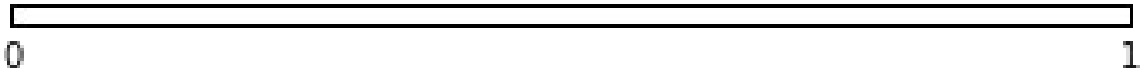
$$C_2^5 \cdot (0.5)^5 = 0.3125$$

Hands-on Session

- Please go to the **"The Binomial Distribution Object"** in the lecture code.

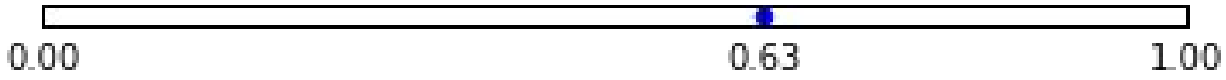
Normal Distributions

- Before our discussion on the normal distribution, let's consider an essential difference from a discrete distribution (such as binomial distribution) to a continuous distribution (what a normal distribution represents).
  - Consider an [0, 1] interval:



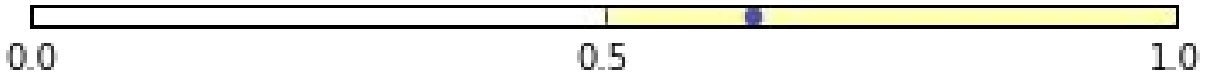
Normal Distributions

- Selecting an arbitrary point from the interval, assume every point is **equally likely** to be selected.
  - What is the "probability" that we get the blue point (x=0.63) below?



Normal Distributions

- Let's consider the process below:
- Cut the interval half.
  - The probability of obtaining a point from each interval should be 50%.



- Since the blue point is contained in one of the intervals, the probability of it should be less than 50%.

Normal Distributions

- Cut the interval into four pieces.
  - The probability of obtaining a point from each interval should be 25%.

- Since the blue point is contained in one of the intervals, the probability of it should be less than 25%.

Normal Distributions

- Cut the interval into ten pieces.
  - The probability of obtaining a point from each interval should be 10%.

- Since the blue point is contained in one of the intervals, the probability of it should be less than 10%.

Normal Distributions

- Continue this process, it's easy to see that the probability of obtaining the blue point is **zero**.
  - In fact, the probability of obtaining any particular point in the interval is **zero**.
- Therefore, for a continuous distribution we talk about the probability of **getting some number from a particular interval**, instead of getting a particular number. Below we demonstrate how this is done with **normal** distribution.

Normal Distributions: The Probability Density Function

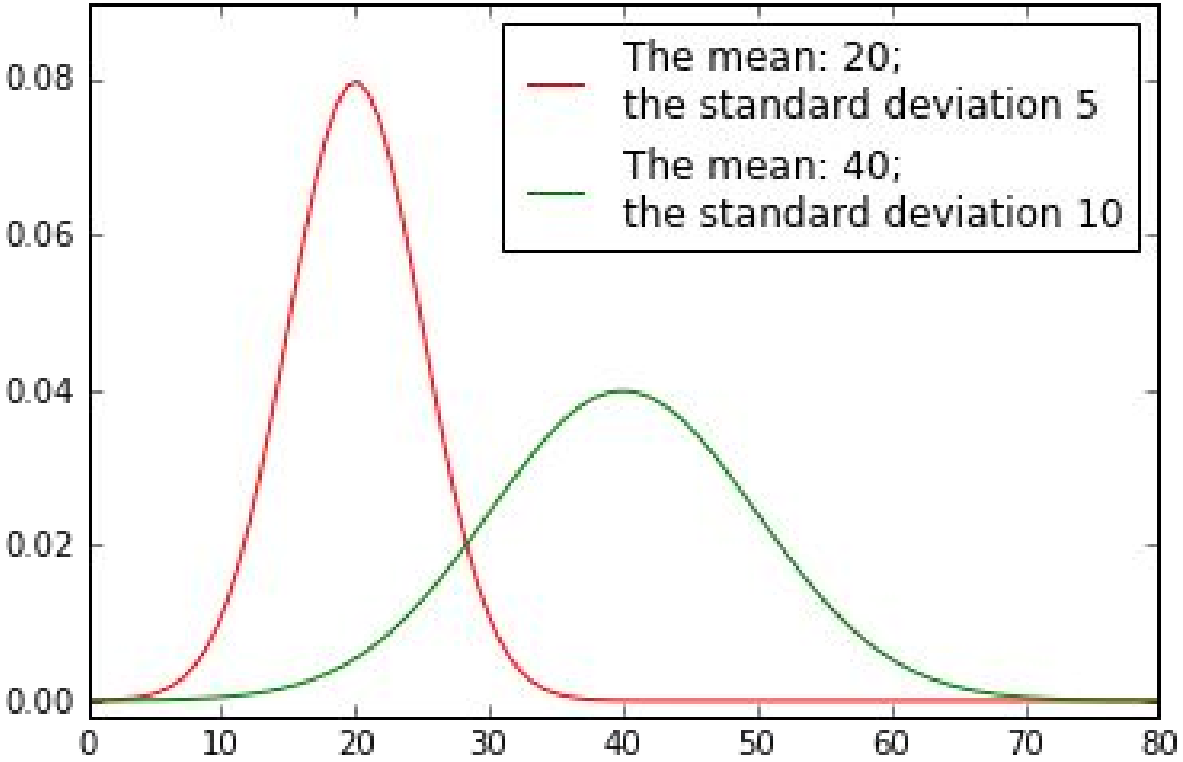
- The most common way to represent a normal distribution is by its probability density function (**pdf**). We will justify the name "density".
- The pdf of a normal distribution if often denoted by:

$$N(\mu,\sigma)(x) = \frac{1}{\sqrt{2\pi}\sigma} exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

where μ is the mean and σ is the standard deviation.

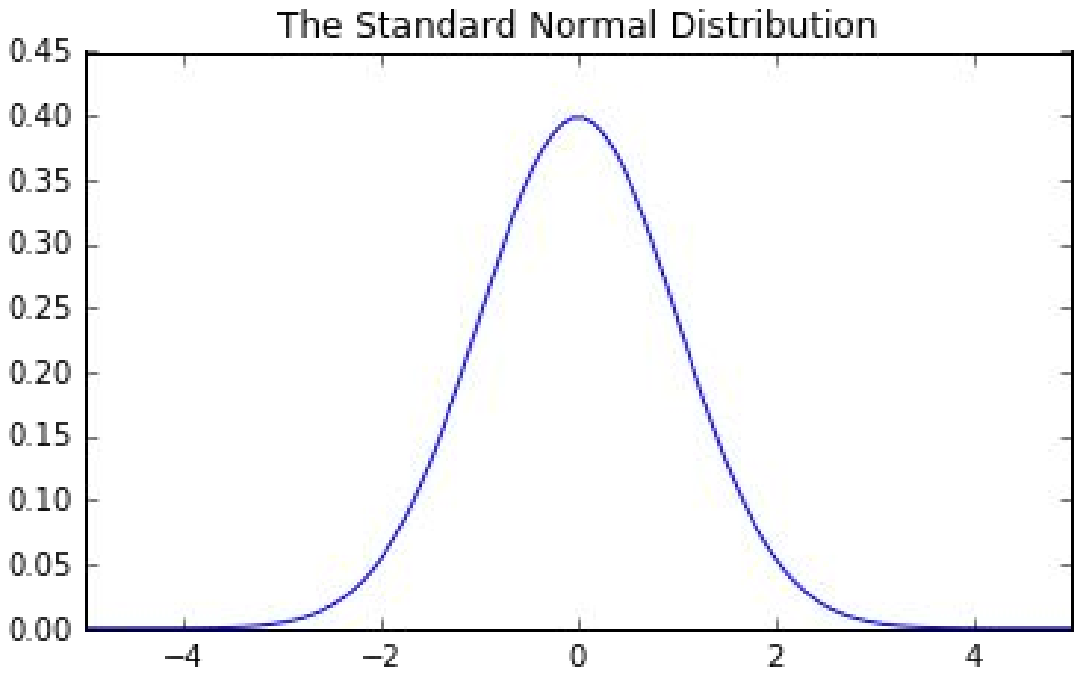
Normal Distributions: The Probability Density Function

- Plot the pdf we obtain the famous bell curve:



Normal Distributions: The Probability Density Function

- The normal distribution is completely determined by its mean and the standard deviation. The normal distribution with $\mu = 0$ and $\sigma = 1$ is called the **standard normal distribution**.

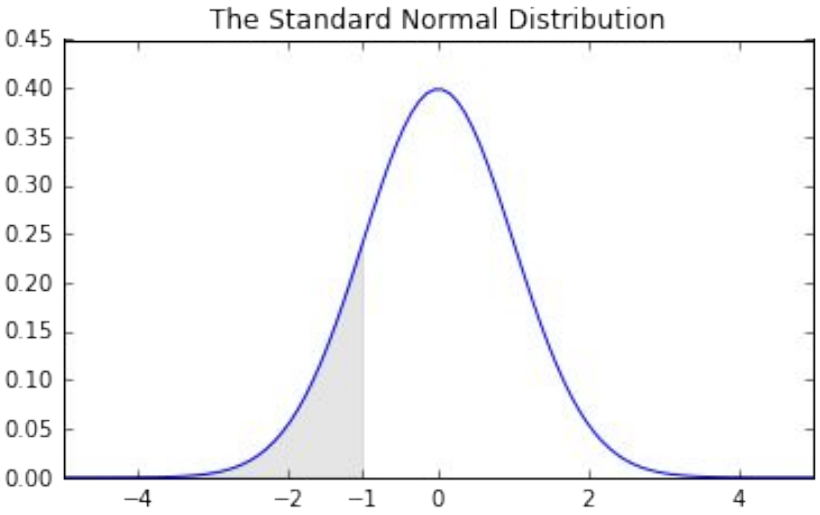


Normal Distributions: The Cumulative Density Function

- The probability density is important of its own right, but probability can be more intuitive.
- For an interval, the probability of the occurrence of any number in it is **the area above the interval and beneath the pdf curve**.

Normal Distributions: The Cumulative Density Function

- The probability of obtaining a number smaller than -2 is the area below:

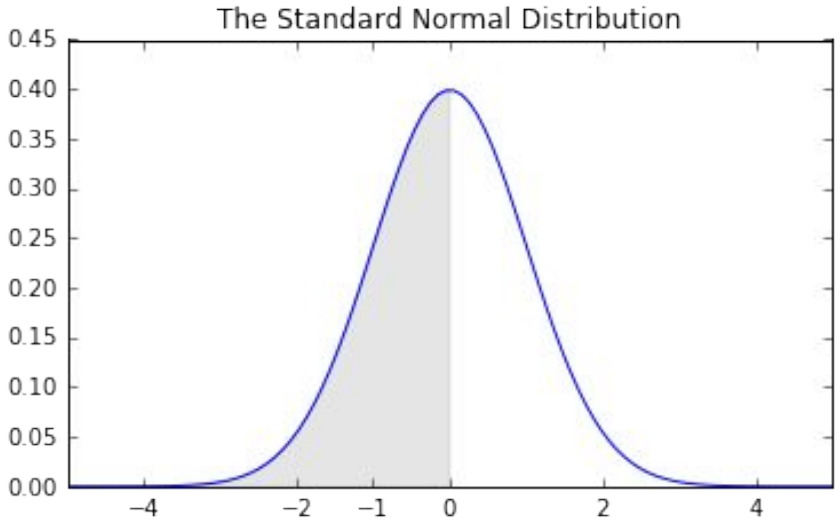


- Equivalently, this can be represented by integral:

$$\int_{-\infty}^{-1} N(0,1)(t)dt$$

Normal Distributions: The Cumulative Density Function

- The probability of obtaining a number smaller than 0 is the area below:



- Equivalently, this can be represented by integral:

$$\int_{-\infty}^{0} N(0,1)(t)dt$$

Normal Distributions: The Cumulative Density Function

- Therefore, the **cumulative density function** is defined as:

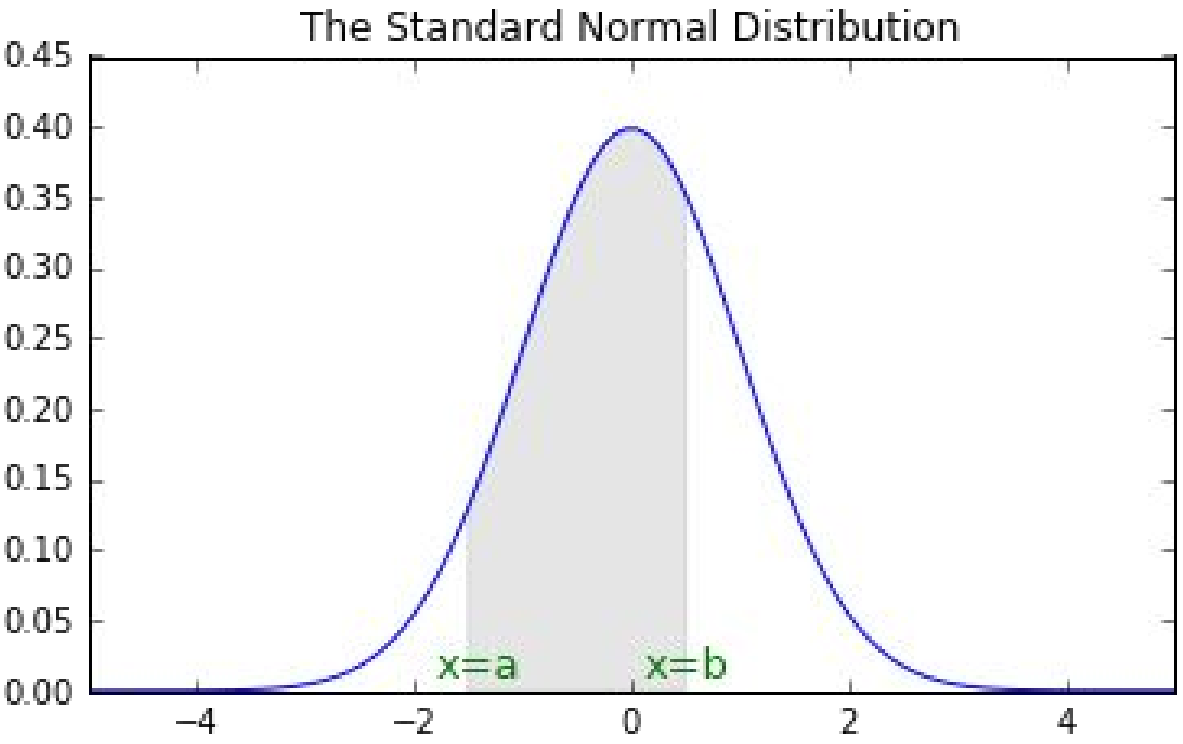
$$\text{cdf}(x) = \int_{-\infty}^{x} N(0,1)(t)dt$$

- Once we understand the cdf function, then the probability of obtaining a value from an arbitrary interval [a, b] is:

$$\text{cdf}(b) - \text{cdf}(a) = \int_{a}^{b} N(0,1)(t)dt$$

Normal Distributions: The Cumulative Density Function

- Or we can visualize this by:



Normal Distributions: The Cumulative Density Function

- To summarise, the normal distribution (as a continuous distribution) are often characterized by
  - the cumulative density function, which assigns a probability to each interval.
  - the probability density function, the area under whose curve assigns a probability to the corresponding interval. More precisely, pdf is actually the derivative of cdf. And since cdf returns probability, pdf returns the probability density.
- **Remark** Even though the probability density is not probability, it does indicate **likelihood**. That is to say, the values with higher probability density are more likely to occur. In fact, in many machine learning algorithm, the likelihood functions are constructed from pdf.

Hands-on Session

- Please go to the **"The Normal Distribution Object"** in the lecture code.