01 NumPy

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called axes.

In [1]:
```
import numpy as np
```

1. Array Creation

- np.array()
- np.zeros()
- np.ones()
- np.empty()
- np.eye()
- np.asarray()
- numpy.arange()
- np.linspace()

1.1 np.array()

Create an array.

- np.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

In [2]:
```
a = np.array([1,2,3,4,5,6]) # create a one-dimensional array
print("one-dimensional array：", a)
```

```
one-dimensional array： [1 2 3 4 5 6]
```

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

In [3]:
```
b = np.array([[1,2,3],[4,5,6]]) # create a multidimensional array
print("multidimensional array：" ,b)
```

```
multidimensional array： [[1 2 3]
 [4 5 6]]
```

1.2 np.zeros(), np.ones(), np.empty()

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64. np.zeros(shape, dtype=float, order='C')

- np.zeros(shape, dtype=float, order='C')
- np.ones(shape, dtype=None, order='C')
- np.empty(shape, dtype=float, order='C')

In [4]:
```
c = np.zeros([2,2])
print("array full of zeros:",c)
```

```
array full of zeros: [[0. 0.]
 [0. 0.]]
```

In [5]:
```
d = np.ones([2,2])
print("array full of ones:",d)
```

```
array full of ones: [[1. 1.]
 [1. 1.]]
```

In [6]:
```
e = np.empty([3,4])
print("array whose initial content is random: ",e)
```

```
array whose initial content is random:  [[ 4.45057637e-308  1.78021527e-306  8.
45549797e-307  1.37962049e-306]
 [ 1.11260619e-306  1.78010255e-306  9.79054228e-307  4.45057637e-308]
 [ 8.45596650e-307  9.34602321e-307  4.94065646e-322 -7.06293859e-311]]
```

1.3 np.eye()

Return a 2-D array with ones on the diagonal and zeros elsewhere.

- np.eye(N, M=None, k=0, dtype=float, order='C')

In [7]:
```
print(np.eye(5))
```

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

1.4 np.asarray()

Convert the input to an array.

- np.asarray(a, dtype = None, order = None)

In [8]:
```
e = np.asarray([1,2,3,4,5,6])
print("one-dimensional array:", e)
```

```
one-dimensional array: [1 2 3 4 5 6]
```

1.5 the difference between array() and asarray()

The main difference is that array (by default) will make a copy of the object, while asarray will not unless necessary.

In [9]:
```
a = np.array([1, 2], dtype=np.float32)

print("data not changed: ",np.array(a, dtype=np.float32) is a)
print("data changed:",np.array(a, dtype=np.float64) is a)
print("data not changed: ",np.asarray(a, dtype=np.float32) is a)
print("data changed:",np.asarray(a, dtype=np.float64) is a)
```

```
data not changed:  False
data changed: False
data not changed:  True
data changed: False
```

1.6 np.arange()

To create sequences of numbers, NumPy provides the arange function which is analogous to the Python built-in range, but returns an array.

- numpy.arange([start,]stop, [step,]dtype=None)
  - start: number, optional; Start of interval. The interval includes this value. The default start value is 0.
  - stop: number,End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of out.

- step: number, optional; Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] - out[i]. The default step size is 1. If step is specified as a position argument, start must also be given.

In [10]:
```
f = np.arange(10,20,2)
print("sequences of numbers in array:",f)
```

```
sequences of numbers in array: [10 12 14 16 18]
```

1.7 np.linspace()

Return evenly spaced numbers over a specified interval.

Returns num evenly spaced samples, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.

- np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

In [11]:
```
g = np.linspace(0,99,10)
print(g)
```

```
[ 0. 11. 22. 33. 44. 55. 66. 77. 88. 99.]
```

2. Important attributes of an ndarray object

- ndarray.ndim
  - the number of axes (dimensions) of the array.
- ndarray.shape
  - the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
- ndarray.size
  - the total number of elements of the array. This is equal to the product of the elements of shape.
- ndarray.dtype
  - an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- ndarray.itemsize
  - the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.
- ndarray.data

- the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

In [12]:
```
a = np.array([[1,2],[3,4],[5,6]])
a.shape
```

Out[12]:
```
(3, 2)
```

In [13]:
```
a.size
```

Out[13]:
```
6
```

In [14]:
```
a.ndim
```

Out[14]:
```
2
```

In [15]:
```
a.dtype
```

Out[15]:
```
dtype('int32')
```

3. Indexing, Slicing and Iterating

ndrrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

In [16]:
```
# Indexing
a = np.array([[1, 2], [3, 4]])
b = a[0][1]
print('a:',a)
print('b:',b)
```

```
a: [[1 2]
 [3 4]]
b: 2
```

In [17]:
```
# Slicing
a = np.arange(10)
print(a[2:8:2])
print(a[2:8])
print(a[2])
print(a[2:])
```

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

4. Change the shape of an array

Gives a new shape to an array without changing its data.

- numpy.reshape(a, newshape, order='C')

In [18]:
```
a = np.arange(6)
b = a.reshape(2,3)
print(a)
print(b)
```

```
[0 1 2 3 4 5]
[[0 1 2]
 [3 4 5]]
```

5. Broadcasting

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape,

In [19]:
```
a = np.array([1,2,3,4])
b = np.array([1,2,3,4])
c = a * b
print(c)
```

```
[ 1  4  9 16]
```

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

- they are equal, or
- one of them is 1

If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Arrays do not need to have the same number of dimensions. For example, if you have a 256x256x3 array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values.

In [20]:
```
a = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
b = np.array([1, 2, 3])
print(a.shape)
print(b.shape)
```

```
(3, 3)
(3,)
```

In [21]:
```
np.array([b,b,b])
```

Out[21]:
```
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

In [22]:
```
c = a + b
print("a:\n",a)
print("broadcasting:\n",c)
```

```
a:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
broadcasting:
 [[ 2  4  6]
 [ 5  7  9]
 [ 8 10 12]]
```

In [23]:
```
import numpy as np
```

6. NumPy common functions

6.1 ndarray.flat()

A 1-D iterator over the array.

This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

In [24]:
```
a = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
print(a)
print(a.flat[:])
```

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

6.2 ndarray.transpose()

Returns a view of the array with axes transposed.

In [25]:
```
a = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
print(np.transpose(a))
```

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

6.3 numpy.concatenate((a1, a2, ...), axis=0, out=None)

Join a sequence of arrays along an existing axis.

- a1, a2, …: The arrays must have the same shape, except in the dimension corresponding to axis (the first, by default).
- axis:The axis along which the arrays will be joined. If axis is None, arrays are flattened before use. Default is 0.

In [26]:
```
a = np.arange(6).reshape(2,3)
b = np.arange(7,13).reshape(2,3)
print(a)
print(b)
```

```
[[0 1 2]
 [3 4 5]]
[[ 7  8  9]
 [10 11 12]]
```

In [27]:
```
print(np.concatenate((a,b), axis = 0))
print(np.concatenate((a,b), axis = 1))
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 7  8  9]
 [10 11 12]]
[[ 0  1  2  7  8  9]
 [ 3  4  5 10 11 12]]
```

In [28]:
```
a = np.arange(12).reshape(2,2,3)
b=np.transpose(a,(2,1,0))
print(a)
print(b)
```

```
[[[ 0  1  2]
  [ 3  4  5]]

 [[ 6  7  8]
  [ 9 10 11]]]
[[[ 0  6]
  [ 3  9]]

 [[ 1  7]
  [ 4 10]]

 [[ 2  8]
  [ 5 11]]]
```

6.4 numpy.stack(arrays, axis=0, out=None)：

Join a sequence of arrays along a new axis.

The axis parameter specifies the index of the new axis in the dimensions of the result. For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension.

In [29]:
```
a = np.array([1, 2])
b = np.array([3, 4])
c = np.stack((a,b),0)
d = np.stack((a,b),1)

print('c\n',c)
print('d\n',d)
```

```
c
 [[1 2]
 [3 4]]
d
 [[1 3]
 [2 4]]
```

6.5 numpy.append(arr, values, axis=None)

Append values to the end of an array.

In [30]:
```
a = np.array([[1,2,3],[4,5,6]])
b = np.append(a,[[7,8,9]],axis=0)
c = np.append(a,[[7,8,9]])
print('b\n',b)
print('c\n',c)
```

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

6.6 numpy.insert(arr, obj, values, axis=None)

Insert values along the given axis before the given indices.

In [31]:
```
a = np.arange(6).reshape(2,3)
b = np.insert(a,2,[7,8],axis=1)
c = np.insert(a,2,[7,8])
print('a=',a)
print('b=',b)
print('c=',c)
```

```
a= [[0 1 2]
 [3 4 5]]
b= [[0 1 7 2]
 [3 4 8 5]]
c= [0 1 7 8 2 3 4 5]
```

6.7 numpy.delete(arr, obj, axis=None)

Return a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by arr[obj].

In [32]:
```
a = np.arange(6).reshape(2,3)
b = np.delete(a,1,axis=1)
c = np.delete(a,1)
print('a=',a)
print('b=',b)
print('c=',c)
```

```
a= [[0 1 2]
 [3 4 5]]
b= [[0 2]
 [3 5]]
c= [0 2 3 4 5]
```

7. Matrix

This module contains all functions in the numpy namespace, with the following replacement functions that return matrices instead of ndarrays.

- empty(shape[, dtype, order])
  - Return a new matrix of given shape and type, without initializing entries.
- zeros(shape[, dtype, order])
  - Return a matrix of given shape and type, filled with zeros.
- ones(shape[, dtype, order])
  - Matrix of ones.
- eye(n[, M, k, dtype, order])
  - Return a matrix with ones on the diagonal and zeros elsewhere.
- identity(n[, dtype])
  - Returns the square identity matrix of given size.
- repmat(a, m, n)
  - Repeat a 0-D to 2-D array or matrix MxN times.
- rand(*args)
  - Return a matrix of random values with given shape.
- randn(*args)
  - Return a random matrix with data from the “standard normal” distribution.

In [33]:
```
import numpy.matlib
```

7.1 creat a matrix by np.matlib.zeros() and np.matlib.ones()

In [34]:
```
a=np.matlib.zeros((2,2)) # matrix full of zeros
b=np.matlib.ones((2,2)) # matrix full of ones
print('a=',a)
print('b=',b)
```

```
a= [[0. 0.]
 [0. 0.]]
b= [[1. 1.]
 [1. 1.]]
```

7.2 creat a diagonal matrix by np.matlib.eye()

In [35]:
```
c=np.matlib.eye(3)
print('c=',c)
```

```
c= [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

7.3 transfer between array and matrix

In [36]:
```
# transfer an array to a matrix
d = np.array([[1,2],[3,4]])
d = np.mat(a)
print(d)
print(type(d))

# transfer a matrix to an array
a=np.mat([[1,2,3],[4,5,6]])
b=a.getA()
print('matrix a=',a)
print(type(a))
print('array b=',b)
print(type(b))
```

```
[[0. 0.]
 [0. 0.]]
<class 'numpy.matrix'>
matrix a= [[1 2 3]
 [4 5 6]]
<class 'numpy.matrix'>
array b= [[1 2 3]
 [4 5 6]]
<class 'numpy.ndarray'>
```

7.4 randomly create a matrix by np.matlib.rand()

In [37]:
```
a=np.matlib.rand(3,3)
print('a=',a)
```

```
a= [[0.07080801 0.9544403  0.58034575]
 [0.92294246 0.08798037 0.25544409]
 [0.00760968 0.32743808 0.79406869]]
```

7.7 matrix operations

matrix.dot(b, out=None) / numpy.dot(a, b, out=None)

Dot product of two arrays.

Dot product of two arrays. Specifically,

- If both a and b are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both a and b are 2-D arrays, it is matrix multiplication, but using matmul or a @ b is preferred.
- If either a or b is 0-D (scalar), it is equivalent to multiply and using numpy.multiply(a, b) or a * b is preferred.
- If a is an N-D array and b is a 1-D array, it is a sum product over the last axis of a and b.
- If a is an N-D array and b is an M-D array (where M>=2), it is a sum product over the last axis of a and the second-to-last axis of b:
  - dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

In [38]:
```
a = np.array([[1, 0], [0, 1]])
b = np.array([[1, 1], [2, 2]])
print(np.dot(a, b)) #  matrix multiplication
print(a@b) #  matrix multiplication
```

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

multiply

In [39]:
```
print(np.multiply(a,b)) # equivalent to multiply
print(a*b) # equivalent to multiply
```

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

7.8 Matrix Inverse

7.5 mat/matrix/asmatrix:

Functions that are also in the numpy namespace and return matrices

- mat(data[, dtype])
  - Interpret the input as a matrix.
- matrix(data[, dtype, copy])
  - Note: It is no longer recommended to use this class, even for linear
- asmatrix(data[, dtype])
  - Interpret the input as a matrix.

In [40]:
```
a=np.array([1,2,3])
b=np.mat(a)
d=np.asmatrix(a)
c=np.matrix(a)
print('array a=',a)
print('matrix b=',b)
print('matrix c=',c)
print('matrix d=',d)
```

```
array a= [1 2 3]
matrix b= [[1 2 3]]
matrix c= [[1 2 3]]
matrix d= [[1 2 3]]
```

In [41]:
```
a=np.matrix([[2,0,0],
             [0,1,0],
             [0,0,2]])
b=a.I
print('matrix a=',a)
print('matrix b=',b)
```

```
matrix a= [[2 0 0]
 [0 1 0]
 [0 0 2]]
matrix b= [[0.5 0.  0. ]
 [0.  1.  0. ]
 [0.  0.  0.5]]
```

7.9 Determinant of a Matrix

In [42]:
```
a=np.matrix([[1,2],
             [3,4]])
b=np.linalg.det(a)
print('matrix a=',a)
print(' b=',b)
```

```
matrix a= [[1 2]
 [3 4]]
 b= -2.0000000000000004
```

7.10 sum, max, min

In [43]:
```
a=np.matrix([[1,2],
             [3,4]])
b=a.sum(axis=0)       # sum by columns
c=a.sum(axis=1)       # sum by raws
print('matrix a=',a)
print(' b=',b)
print(' c=',c)
```

```
matrix a= [[1 2]
 [3 4]]
 b= [[4 6]]
 c= [[3]
 [7]]
```

In [44]:
```
d=a.max()
e=a.min()
print(' d=',d)
print(' e=',e)
```

```
 d= 4
 e= 1
```

8. Random sampling

8.1 Simple random data

- rand(d0, d1, …, dn) Random values ([0.0, 1.0)) in a given shape.
- randn(d0, d1, …, dn) Return a sample (or samples) from the “standard normal” distribution.
- randint(low[, high, size, dtype]) Return random integers from low (inclusive) to high (exclusive).

In [45]:
```
r1=np.random.rand(2,2)
r2=np.random.randn(2,2)
r3=np.random.randint(0,5)

print('r1=','\n',r1)
print('r2=','\n',r2)
print('r3=','\n',r3)
```

```
r1=
 [[0.66916096 0.14652676]
 [0.75969953 0.21959318]]
r2=
 [[ 0.86393278 -0.00214033]
 [-0.53927724 -0.93445815]]
r3=
 4
```

- numpy.random.choice(a, size=None, replace=True, p=None)
  - a : If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a was np.arange(n)
  - size : Output shape.
  - replace : boolean, optional; Whether the sample is with or without replacement
  - p : 1-D array-like, optional; The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

In [46]:
```
r3=np.random.choice([1,2,3,4,5], (2,2), p=[0.1, 0, 0.3, 0.6, 0])
print(r3)
```

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

8.2 Permutations

- shuffle(x) Modify a sequence in-place by shuffling its contents.
- permutation(x) Randomly permute a sequence, or return a permuted range.

In [47]:
```
a=np.array([1,2,3,4,5])
np.random.shuffle(a)
print('a=','\n',a)
b=np.random.permutation([1,2,3,4,5])

print('a=','\n',a)
print('b=','\n',b)
```

```
a=
 [4 5 3 1 2]
a=
 [4 5 3 1 2]
b=
 [5 1 2 4 3]
```

8.3 Distributions

- normal([loc, scale, size]) Draw random samples from a normal (Gaussian) distribution.
  - loc : Mean (“centre”) of the distribution.
  - scale : Standard deviation (spread or “width”) of the distribution.
  - size : Output shape.
- uniform([low, high, size]) Draw samples from a uniform distribution.
  - Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by uniform.
  - low : Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.
  - high : Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.
  - size : Output shape.
- poisson([lam, size]) Draw samples from a Poisson distribution.
  - lam : Expectation of interval, should be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.
  - size : Output shape.

In [48]:
```
r1=np.random.normal(0,0.1,5)
r2=np.random.uniform(0,5,2)
r3=np.random.poisson(5,2)

print('r1=','\n',r1)
print('r2=','\n',r2)
print('r3=','\n',r3)
```

```
r1=
 [-0.05121693 -0.0019207  -0.04696195 -0.02479263 -0.03872183]
r2=
 [3.95478459 2.68926268]
r3=
 [5 3]
```

9. Common mathematical functions

9.1 Trigonometric functions

- sin(x, /[, out, where, casting, order, …]) Trigonometric sine, element-wise.
- cos(x, /[, out, where, casting, order, …]) Cosine element-wise.

In [49]:
```
a=np.array([0,30,45,60,90])
b=np.sin(a*np.pi/180)
c=np.cos(a*np.pi/180)
print('b=',b)
print('c=',c)
```

```
b= [0.         0.5        0.70710678 0.8660254  1.        ]
c= [1.00000000e+00 8.66025404e-01 7.07106781e-01 5.00000000e-01
 6.12323400e-17]
```

9.2 Rounding

- around(a[, decimals, out]) Evenly round to the given number of decimals.
- floor(x, /[, out, where, casting, order, …]) Return the floor of the input, element-wise.
- ceil(x, /[, out, where, casting, order, …]) Return the ceiling of the input, element-wise.

In [50]:
```
a=np.array([1.0,1.5,2.0,2.55])
b=np.around(a)
c=np.around(a,decimals=1)
print('b=',b)
print('c=',c)
```

```
b= [1. 2. 2. 3.]
c= [1.  1.5 2.  2.6]
```

In [51]:
```
a=np.array([1.0,1.5,2.0,2.55])
d=np.floor(a)
e=np.ceil(a)
print('d=',d)
print('e=',e)
```

```
d= [1. 1. 2. 2.]
e= [1. 2. 2. 3.]
```

9.3 Arithmetic operations

In [52]:
```
a=np.array([1,2,3,4])
b=np.array([4,3,2,1])
c=np.add(a,b)
d=np.subtract(a,b)
e=np.multiply(a,b)
f=np.divide(a,b)
g=np.mod(a,b)
h=np.power(a,b)
print('c=',c)
print('d=',d)
print('e=',e)
print('f=',f)
print('g=',g)
print('h=',h)
```

```
c= [5 5 5 5]
d= [-3 -1  1  3]
e= [4 6 6 4]
f= [0.25       0.66666667 1.5        4.        ]
g= [1 2 1 0]
h= [1 8 9 4]
```

9.4 Statistical function

- numpy.amin(a, axis=None, out=None, keepdims=, initial=)
  - Return the minimum of an array or minimum along an axis.
- numpy.amax(a, axis=None, out=None, keepdims=, initial=)
  - Return the maximum of an array or maximum along an axis.

- numpy.median(a, axis=None, out=None, overwrite_input=False, keepdims=False)
  - Compute the median along the specified axis.Returns the median of the array elements.
- numpy.mean(a, axis=None, dtype=None, out=None, keepdims=)
  - Compute the arithmetic mean along the specified axis.

In [53]:
```
a=np.array([[6, 7, 2], [0, 1, 5]])
b=np.amin(a,0)
c=np.amax(a,1)
d=np.median(a)
e=np.mean(a)
print('a=',a)
print('b=',b)
print('c=',c)
print('d=',d)
print('e=',e)
```

```
a= [[6 7 2]
 [0 1 5]]
b= [0 1 2]
c= [7 5]
d= 3.5
e= 3.5
```

9.5 Sort function

- numpy.sort(a, axis=-1, kind='quicksort', order=None)

Return a sorted copy of an array.

In [54]:
```
a=np.array([[3,5,1],[2,8,7]])
b=np.sort(a)
c=np.sort(a,axis=0)
print(b)
print(c)
```

```
[[1 3 5]
 [2 7 8]]
[[2 5 1]
 [3 8 7]]
```

02 Pandas

pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

In [55]:
```
import pandas as pd
import numpy as np
```

1. Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.

1.1 create a Series

- pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)

In [56]:
```
s = pd.Series([1,3,5,np.nan,6,8]) # np.nan: Not a Number
print(s) # NaN (not a number) is the standard missing data marker used in pandas.
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

1.2 create a Series from an array

In [57]:
```
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print(s)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

1.3 create a Series from a dict

When the data is a dict, and an index is not passed, the Series index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and Pandas version >= 0.23.

In [58]:
```
data1 = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data1)
print(s)
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

2. DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used Pandas object.

2.1 create a DataFrame from dict of Series or dicts

In [59]:
```
# create random numbers as values
df = pd.DataFrame(np.random.randn(7,4), index=[2,3,4,5,6,7,8]
                  , columns=list('ABCD'))
df
```

Out[59]:

	A	B	C	D
2	1.051711	0.922282	1.159198	0.442173
3	0.408800	-1.572626	0.512048	-0.876617
4	-1.073903	-0.859808	-1.426011	-0.241904
5	0.432528	-1.502886	-0.058611	0.134841
6	0.468681	-0.871051	0.295950	-0.707793
7	-0.760660	-1.292461	0.113625	-0.003118
8	0.252282	-1.309093	1.594634	-1.465940

2.2 create a DataFrame from dict of Series or dicts

In [60]:
```
df1 = pd.DataFrame({ 'A' : 1.,
                     'B' : pd.Timestamp('20200102'),
                     'C' : pd.Series(1,index=list(range(4)),dtype='float32'),
                     'D' : np.array([3] * 4,dtype='int32'),
                    # Categoricals can only take on only a limited, and usually f
                     'E' : pd.Categorical(["test","train","test","train"]),
                     'F' : 'foo' })
df1
```

Out[60]:

	A	B	C	D	E	F
0	1.0	2020-01-02	1.0	3	test	foo
1	1.0	2020-01-02	1.0	3	train	foo
2	1.0	2020-01-02	1.0	3	test	foo
3	1.0	2020-01-02	1.0	3	train	foo

3. Essential basic functionality

3.1 Head and tail

To view a small sample of a Series or DataFrame object, use the head() and tail() methods. The default number of elements to display is five, but you may pass a custom number.

In [61]:
```
data2 = np.arange(300).reshape(60,5)
df2 = pd.DataFrame(data2)
```

3.1.1 DataFrame.head(n=5)：

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

In [62]:
```
df2.head(3)
```

Out[62]:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14

3.1.2 DataFrame.tail(n=5)：

This function returns last n rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

In [63]:
```
df2.tail(3)
```

Out[63]:

	0	1	2	3	4
57	285	286	287	288	289
58	290	291	292	293	294
59	295	296	297	298	299

3.2 Indexing and selecting data

3.2.1 index, columns and values

In [64]:
```
# create random numbers as values
df = pd.DataFrame(np.random.randn(7,4), index=['a','b','c','d','e','f','g']
                  , columns=list('ABCD'))
df.index
```

Out[64]:
```
Index(['a', 'b', 'c', 'd', 'e', 'f', 'g'], dtype='object')
```

In [65]:
```
df.columns
```

Out[65]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

In [66]:
```
df.values
```

Out[66]:
```
array([[ 0.18148929, -0.674277  ,  0.32477614,  1.81846219],
       [-0.10318377,  1.24394995, -0.66926691,  0.11292757],
       [ 0.67731365, -0.70240702,  1.44673689,  1.7184988 ],
       [-0.5819594 , -1.69843186,  0.103455  , -0.34111442],
       [-0.56979182,  0.12591963,  0.60501501,  0.73018821],
       [-1.00987306, -1.23148145, -2.23697222,  0.08859807],
       [ 1.76866222, -0.42751919,  0.18694193, -0.76838628]])
```

3.2.2 loc, iloc

- .loc is primarily label based, but may also be used with a boolean array. .loc will raise KeyError when the items are not found.
- .iloc is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array. .iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy slice semantics).

In [67]:
```
df1 = pd.DataFrame(np.random.randn(6, 4),
                   index=list('abcdef'),
                   columns=list('ABCD'))
df1
```

Out[67]:

	A	B	C	D
a	-0.250412	0.291404	0.766375	0.902152
b	-0.231931	1.139454	0.495638	0.281947
c	1.423526	0.792302	-0.597939	-1.057672
d	-1.194916	-1.625338	0.035786	-0.797056
e	-1.873345	0.112021	0.108004	-1.483627
f	-0.392886	-0.531650	-0.911324	2.378760

In [68]:
```
df1.loc[['a', 'b', 'd'], :]
```

Out[68]:

	A	B	C	D
a	-0.250412	0.291404	0.766375	0.902152
b	-0.231931	1.139454	0.495638	0.281947
d	-1.194916	-1.625338	0.035786	-0.797056

In [69]:
```
df1.iloc[[0, 1, 3], :]
```

Out[69]:

	A	B	C	D
a	-0.250412	0.291404	0.766375	0.902152
b	-0.231931	1.139454	0.495638	0.281947
d	-1.194916	-1.625338	0.035786	-0.797056

3.3 Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on Series, DataFrame. Most of these are aggregations (hence producing a lower-dimensional result) like sum(), mean(), and quantile(), but some of them, like cumsum() and cumprod(), produce an object of the same size. Generally speaking, these methods take an axis argument, just like ndarray.{sum, std, …}, but the axis can be specified by name or integer:

- Series: no axis argument needed
- DataFrame: “index” (axis=0, default), “columns” (axis=1)

3.3.1 Descriptive statistics functions

- count: Number of non-NA observations
- mean: Mean of values
- std: Bessel-corrected sample standard deviation
- sum: Sum of values
- median: Arithmetic median of values
- min: Minimum
- max: Maximum
- quantile: Sample quantile (value at %)
- abs: Absolute Value
- prod: Product of values
- var: Unbiased variance

In [70]:
```
print(df1.count())
print(df1.mean())
print(df1.std())
print(df1.sum())
```

```
A    6
B    6
C    6
D    6
dtype: int64
A   -0.419994
B    0.029699
C   -0.017243
D    0.037417
dtype: float64
A    1.111410
B    0.994248
C    0.637525
D    1.449040
dtype: float64
A   -2.519964
B    0.178194
C   -0.103461
D    0.224504
dtype: float64
```

3.3.2 Summarizing data: describe

There is a convenient describe() function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course).

In [71]:
```
df2.describe()
```

Out[71]:

	0	1	2	3	4
count	60.000000	60.000000	60.000000	60.000000	60.000000
mean	147.500000	148.500000	149.500000	150.500000	151.500000
std	87.321246	87.321246	87.321246	87.321246	87.321246
min	0.000000	1.000000	2.000000	3.000000	4.000000
25%	73.750000	74.750000	75.750000	76.750000	77.750000
50%	147.500000	148.500000	149.500000	150.500000	151.500000
75%	221.250000	222.250000	223.250000	224.250000	225.250000
max	295.000000	296.000000	297.000000	298.000000	299.000000

4. Dropping labels from an axis

The drop() function. It removes a set of labels from an axis

In [72]:
```
data3 = np.arange(30).reshape(6,5)
df3 = pd.DataFrame(data3,index=['a','b','c','d','e', 'f'] , columns = ['A', 'B',
df3
```

Out[72]:

	A	B	C	D	E
a	0	1	2	3	4
b	5	6	7	8	9
c	10	11	12	13	14
d	15	16	17	18	19
e	20	21	22	23	24
f	25	26	27	28	29

In [73]:
```
a = df3.drop(['a'], axis=0) # axis=0, delete rows
a
```

Out[73]:

	A	B	C	D	E
b	5	6	7	8	9
c	10	11	12	13	14
d	15	16	17	18	19
e	20	21	22	23	24
f	25	26	27	28	29

In [74]:
```
b = df3.drop(['A'], axis=1) # axis=1, delete columns
b
```

Out[74]:

	B	C	D	E
a	1	2	3	4
b	6	7	8	9
c	11	12	13	14
d	16	17	18	19
e	21	22	23	24
f	26	27	28	29

5. DataFrame.append

Append rows of other to the end of caller, returning a new object.Columns in other that are not in the caller are added as new columns.

In [75]:
```
c = b.append(a)
c
```

Out[75]:

	B	C	D	E	A
a	1	2	3	4	NaN
b	6	7	8	9	NaN
c	11	12	13	14	NaN
d	16	17	18	19	NaN
e	21	22	23	24	NaN
f	26	27	28	29	NaN
b	6	7	8	9	5.0
c	11	12	13	14	10.0
d	16	17	18	19	15.0
e	21	22	23	24	20.0
f	26	27	28	29	25.0

6. Reindexing and altering labels

reindex() is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To reindex means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, fill data for missing labels using logic (highly relevant to working with time series data)

In [76]:
```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
s
```

Out[76]:
```
a   -0.293699
b   -0.542152
c    0.619486
d    1.650400
e   -1.884841
dtype: float64
```

In [77]:
```
s.reindex(['e', 'b', 'f', 'd']) # the f label was not contained in the Series and
```

Out[77]:
```
e   -1.884841
b   -0.542152
f         NaN
d    1.650400
dtype: float64
```

In [78]:
```
index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
                   'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
                  index=index)
df
```

Out[78]:

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

In [79]:
```
new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
             'Chrome']
df.reindex(new_index)
```

Out[79]:

	http_status	response_time
Safari	404.0	0.07
Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

7. DataFrame.iteritems

Iterate over (column name, Series) pairs.Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

In [80]:
```
df4 = pd.DataFrame(np.random.randn(4,3),columns=['A','B','C'])
df4
```

Out[80]:

	A	B	C
0	0.709222	-0.451904	0.794877
1	-1.649814	1.218105	0.321418
2	0.891291	0.426845	-0.281538
3	0.339299	0.746237	0.794508

In [81]:
```
i = 1
for s in df4.iteritems():
    print("column: %d value: %s"%(i,s))
    i += 1
```

```
column: 1 value: ('A', 0    0.709222
1   -1.649814
2    0.891291
3    0.339299
Name: A, dtype: float64)
column: 2 value: ('B', 0   -0.451904
1    1.218105
2    0.426845
3    0.746237
Name: B, dtype: float64)
column: 3 value: ('C', 0    0.794877
1    0.321418
2   -0.281538
3    0.794508
Name: C, dtype: float64)
```

8. Statistical functions

8.1 Percent change

Series and DataFrame have a method pct_change() to compute the percent change over a given number of periods (using fill_method to fill NA/null values before computing the percent change).

In [82]:
```
s = pd.Series([1,2,3,4,5,4])
print(s.pct_change())
```

```
0         NaN
1    1.000000
2    0.500000
3    0.333333
4    0.250000
5   -0.200000
dtype: float64
```

8.2 Covariance

Series.cov() can be used to compute covariance between series (excluding missing values).

In [83]:
```
s1 = pd.Series(np.random.randn(1000))
s2 = pd.Series(np.random.randn(1000))
s1.cov(s2)
```

Out[83]:
```
-0.04875415434629192
```

Analogously, DataFrame.cov() to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

In [84]:
```
frame = pd.DataFrame(np.random.randn(1000, 5),
                     columns=['a', 'b', 'c', 'd', 'e'])
frame.cov()
```

Out[84]:

	a	b	c	d	e
a	0.981896	0.002914	-0.018541	0.051356	-0.030724
b	0.002914	0.943477	-0.002104	-0.011866	0.009113
c	-0.018541	-0.002104	0.990995	0.012637	-0.001635
d	0.051356	-0.011866	0.012637	0.967933	-0.031505
e	-0.030724	0.009113	-0.001635	-0.031505	1.071471

9. Sorting

Pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

9.1 By index

The Series.sort_index() and DataFrame.sort_index() methods are used to sort a pandas object by its index levels.

In [85]:
```
df = pd.DataFrame({
                    'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
                    'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
                    'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})

unsorted_df = df.reindex(index=['d', 'a', 'c', 'b'],
                         columns=['three', 'two', 'one'])

unsorted_df
```

Out[85]:

	three	two	one
d	0.902881	-0.867396	NaN
a	NaN	-0.535050	-1.628511
c	0.584258	1.220215	0.863351
b	-1.767471	0.833227	-1.398333

In [86]:
```
unsorted_df.sort_index()
```

Out[86]:

	three	two	one
a	NaN	-0.535050	-1.628511
b	-1.767471	0.833227	-1.398333
c	0.584258	1.220215	0.863351
d	0.902881	-0.867396	NaN

In [87]:
```
unsorted_df.sort_index(ascending=False)
```

Out[87]:

	three	two	one
d	0.902881	-0.867396	NaN
c	0.584258	1.220215	0.863351
b	-1.767471	0.833227	-1.398333
a	NaN	-0.535050	-1.628511

9.2 By values

The Series.sort_values() method is used to sort a Series by its values. The DataFrame.sort_values() method is used to sort a DataFrame by its column or row values. The optional by parameter to DataFrame.sort_values() may used to specify one or more columns to use to determine the sorted order.

In [88]:
```
df1 = pd.DataFrame({'one': [2, 1, 1, 1],
                    'two': [1, 3, 2, 4],
                  'three': [5, 4, 3, 2]})
df1.sort_values(by='two')
```

Out[88]:

	one	two	three
0	2	1	5
2	1	2	3
1	1	3	4
3	1	4	2

10. Working with missing data

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python None will arise and we wish to also consider that “missing” or “not available” or “NA”.

In [89]:
```
df6 = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f','h'],columns=
df6 = df6.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
df6
```

Out[89]:

	one	two	three
a	1.040293	0.675159	-1.511299
b	NaN	NaN	NaN
c	0.034063	1.228435	-0.346130
d	NaN	NaN	NaN
e	-0.096107	0.035208	0.613255
f	-0.604101	0.586558	0.306506
g	NaN	NaN	NaN
h	0.473634	-0.380336	2.430214

10.1 Detect missing values

To make detecting missing values easier (and across different array dtypes), pandas provides the isna() and notna() functions, which are also methods on Series and DataFrame objects.

In [90]:
```
df6['one'].isna()
```

Out[90]:
```
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool
```

10.2 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

- When summing data, NA (missing) values will be treated as zero.
- If the data are all NA, the result will be 0.
- Cumulative methods like cumsum() and cumprod() ignore NA values by default, but preserve them in the resulting arrays. To override this behaviour and include NA values, use skipna=False.

In [91]:
```
df6.sum(axis=1)
```

Out[91]:
```
a    0.204153
b    0.000000
c    0.916369
d    0.000000
e    0.552356
f    0.288963
g    0.000000
h    2.523513
dtype: float64
```

10.3 Filling missing values: fillna

fillna() can “fill in” NA values with non-NA data in a couple of ways, which we illustrate:

In [92]:
```
dff = pd.DataFrame(np.random.randn(10, 3), columns=list('ABC'))
dff.iloc[3:5, 0] = np.nan
dff.iloc[4:6, 1] = np.nan
dff.iloc[5:8, 2] = np.nan
dff
```

Out[92]:

	A	B	C
0	-0.137260	-1.067319	-0.878455
1	0.765933	0.961335	-0.834035
2	1.619094	-0.862543	0.145368
3	NaN	-1.252140	1.251394
4	NaN	NaN	0.625081
5	-1.125327	NaN	NaN
6	-0.847875	1.689168	NaN
7	-2.126438	0.258907	NaN
8	0.677811	-2.076954	1.123412
9	0.346294	-0.899739	-0.349067

In [93]:
```
dff.mean()
```

Out[93]:
```
A   -0.103471
B   -0.406161
C    0.154814
dtype: float64
```

In [94]:
```
# Replace NA with a scalar value
dff.fillna(dff.mean())
```

Out[94]:

	A	B	C
0	-0.137260	-1.067319	-0.878455
1	0.765933	0.961335	-0.834035
2	1.619094	-0.862543	0.145368
3	-0.103471	-1.252140	1.251394
4	-0.103471	-0.406161	0.625081
5	-1.125327	-0.406161	0.154814
6	-0.847875	1.689168	0.154814
7	-2.126438	0.258907	0.154814
8	0.677811	-2.076954	1.123412
9	0.346294	-0.899739	-0.349067

In [95]:
```
dff.mean()
```

Out[95]:
```
A   -0.103471
B   -0.406161
C    0.154814
dtype: float64
```

10.4 Dropping axis labels with missing data: dropna

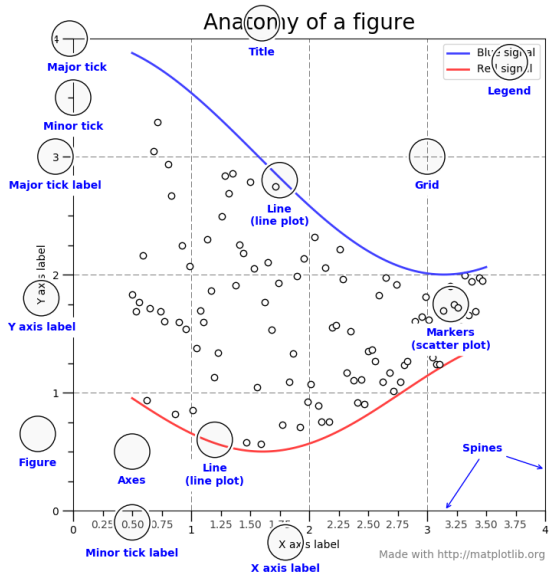
Use dropna() to exclude labels from a data set which refer to missing data.

In [96]:
```
dff.dropna()
```

Out[96]:

	A	B	C
0	-0.137260	-1.067319	-0.878455
1	0.765933	0.961335	-0.834035
2	1.619094	-0.862543	0.145368
8	0.677811	-2.076954	1.123412
9	0.346294	-0.899739	-0.349067

03 Matplotlib

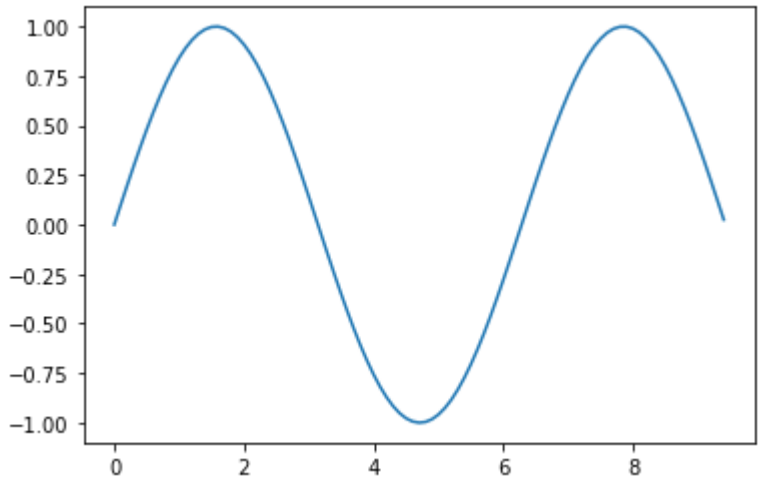


In [97]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

1. Plot

1.1 Plot a graph

In [98]:
```
# generate data for plotting
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)
# plot y versus x as lines and/or markers. *args inclues data, color, format...et
plt.plot(x, y)

# show the graph
plt.show()
```



In [99]:
```
len(x)
```

Out[99]: 95

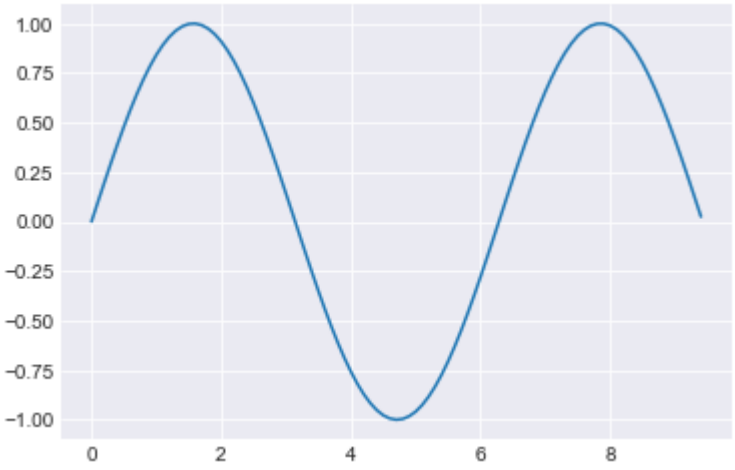
set styles

In [100]:
```
# check available styles
plt.style.available
```

Out[100]:
```
['Solarize_Light2',
 '_classic_test_patch',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark',
 'seaborn-dark-palette',
 'seaborn-darkgrid',
 'seaborn-deep',
 'seaborn-muted',
 'seaborn-notebook',
 'seaborn-paper',
 'seaborn-pastel',
 'seaborn-poster',
 'seaborn-talk',
 'seaborn-ticks',
 'seaborn-white',
 'seaborn-whitegrid',
 'tableau-colorblind10']
```

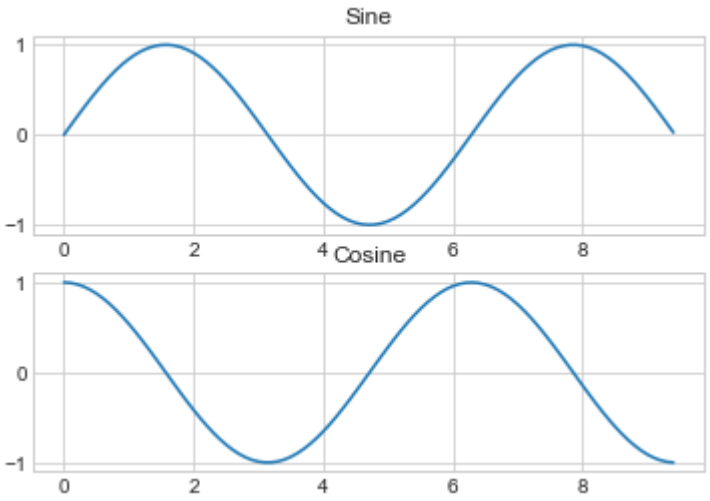
In [101]:
```
# choose style
plt.style.use("seaborn-darkgrid")
```

In [102]:
```
# generate data for plotting
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)
# plot y versus x as lines and/or markers. *args inclues data, color, format...et
plt.plot(x, y)

# show the graph
plt.show()
```



1.2 Combine multiple graphs

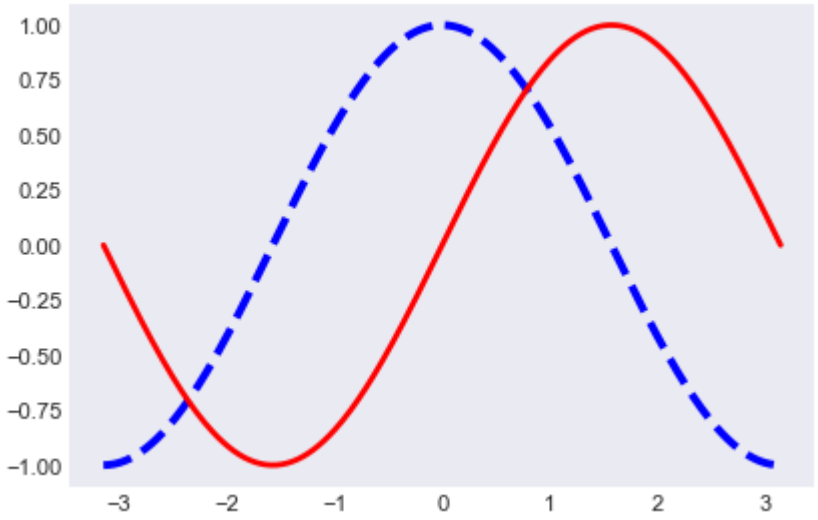
In [103]:
```
# set style
plt.style.use("seaborn-whitegrid")

# generate data for plotting
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# matplotlib.pyplot.subplot(* args, ** kwargs)
# add a subplot to the current figure.*args
# *args can be three integers (nrows, ncols, index),
# or a 3-digit integer. The digits are interpreted as if given separately as thre
plt.subplot(2, 1, 1) # first subplot
plt.plot(x, y_sin) # plot the first subplot
plt.title('Sine') # set title for the first subplot
plt.subplot(2, 1, 2) # second subplot
plt.plot(x, y_cos) # plot second subplot
plt.title('Cosine') # set title for the second subplot
plt.show()
```



1.3 Matplotlib custom numerus formatting like color, marker and linestyle.

In [104]:
```
plt.style.use("seaborn-dark")

# create a 8x6 (inches) figure, dpi means the resolution of the figure (dpi=80, &
plt.figure(figsize=(8,6), dpi=80)


# generate data
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)

# draw a blue curve with customized format.
plt.plot(X, C, color="blue",linewidth=1.0,label="Blue",linestyle="--")
# draw a green curve with cusomized format.
plt.plot(X, S, color="green", linewidth=1.0, label="Green", linestyle="-.")

plt.legend() # place a legend on the axes
plt.xlim(-4.0,4.0) # set the scale of the x-axis
plt.xticks(np.linspace(-4,4,9,endpoint=True)) # set the current tick locations ar
plt.ylim(-1.0,1.0) # set the scale of the y-axis
plt.yticks(np.linspace(-1,1,5,endpoint=True)) # set the current tick locations ar
# plt.savefig("exercice.png",dpi=72) # save figure
plt.show()
```

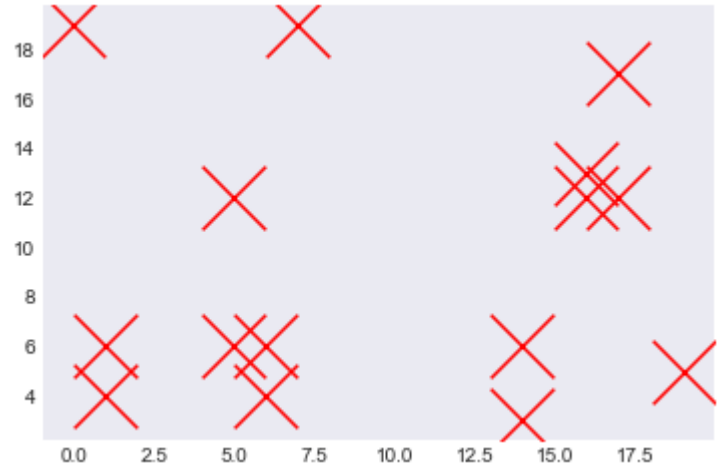
In [105]:
```
# Change color and linewidth：
plt.figure(dpi=80)
plt.plot(X, C, color="blue", linewidth=3.5, linestyle="--")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
plt.show()
```



2. scatter plot

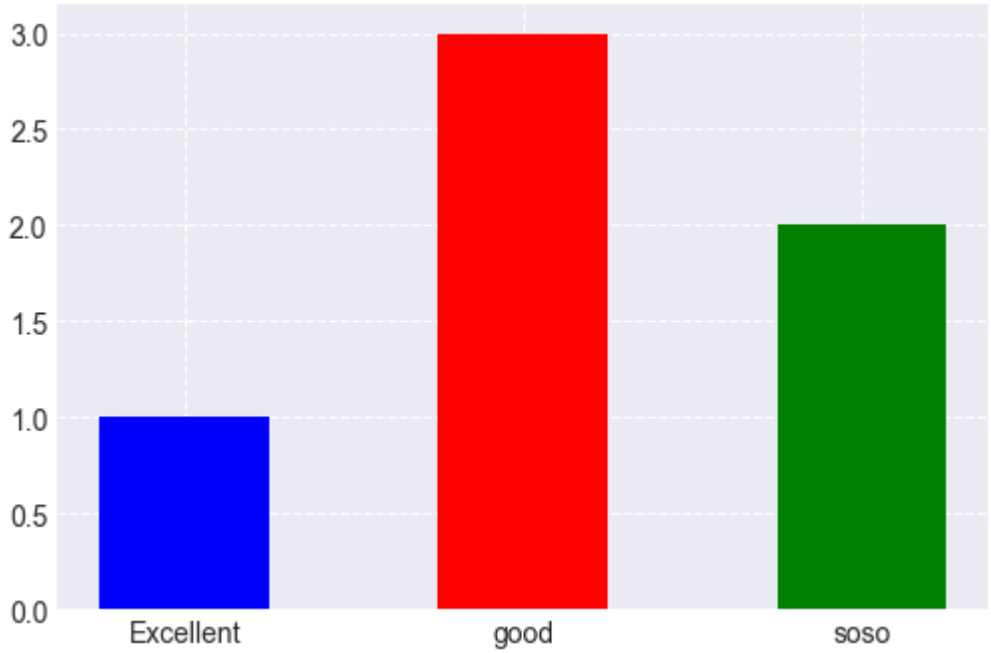
In [106]:
```
# generate data
a = np.random.randint(0,20,15)
b = np.random.randint(0,20,15)
print(a)
print(b)

# matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=No
# A scatter plot of y vs. x with varying marker size and/or color.
plt.scatter(a, b, c='red', s=1000, marker='x')# plot scatter
plt.show()
```

```
[ 1 19  0  6  7 16 14 17 17 16  6 14  5  1  5]
[ 4  5 19  6 19 12  6 12 17 13  4  3  6  6 12]
```



3. Bar plot

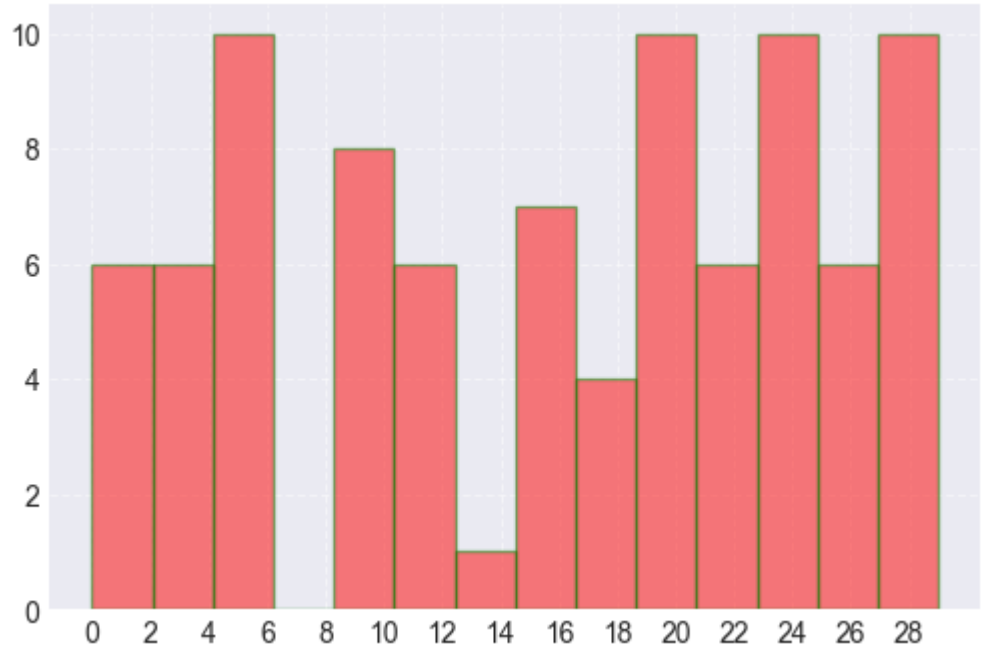
In [107]:
```
level = ['Excellent', 'good', 'soso']
x = range(len(level)) # x-axis
y = [1,3,2] # y-axis
plt.figure(dpi=100) # create figure

# matplotlib.pyplot.bar(x, height, width=0.8, bottom=None, *, align='center', dat
# make a bar plot.
plt.bar(x, y, width=0.5, color=['b','r','g']) # plot figure
plt.xticks(x, level)

# add grid
plt.grid(linestyle="--", alpha=1)
plt.show()
```



4. Histogram

In [108]:
```
# generate data
t = np.random.randint(0,30,90)
print(t)

# create figure
plt.figure(dpi=100)

# set group number (i.e. the range of each bin)

group_num = 14

# matplotlib.pyplot.hist(x, bins=None, range=None, density=False, weights=None, c
# plot a histogram. x indicates the the data. bins indicates the number of bins i
plt.hist(t, facecolor="red", edgecolor="green", bins=group_num, alpha=0.5)
plt.xticks(range(min(t), max(t))[::2])
plt.grid(linestyle="--", alpha=0.5)
plt.show()
```

```
[ 9 19 23 15 22  2 19  5 29  1 17 21 24 26  9 22 20 21 26  3  1 23 12 28
  6 28 21 29 27 12 29 15 15 10 26 16  0 25  5  5 25  5 29 21  5  9 16  3
 20 24 20  1  9 24 19  4  0 17 29 20  4 11  9 10  5 24  3  6  6 23 17 19
 12 20 10 19 13 17  6 23 16 23 29 12 25 29 23  3 12 15]
```



04 SciPy

1. Constants (scipy.constants)

1.1 Mathematical constants

- pi: Pi

1.2 Physical constants

- c: speed of light in vacuum
- speed_of_light: speed of light in vacuum
- h: the Planck constant h
- G: Newtonian constant of gravitation
- electron_mass: electron mass

In [109]:
```
from scipy.constants import *
print(pi)
print(c)
print(speed_of_light)
print(h)
print(G)
print(electron_mass)
```

```
3.141592653589793
299792458.0
299792458.0
6.62607015e-34
6.6743e-11
9.1093837015e-31
```

2. Discrete Fourier transforms (scipy.fft)

2.1 Fast Fourier Transforms (FFTs)

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when n is a power of 2, and the transform is therefore most efficient for these sizes. For poorly factorizable sizes, scipy.fft uses Bluestein's algorithm and so is never worse than O(n log n).

- fft(x[, n, axis, norm, overwrite_x, …]): Compute the 1-D discrete Fourier Transform.
- ifft(x[, n, axis, norm, overwrite_x, …]): Compute the 1-D inverse discrete Fourier Transform.

In [110]:
```
import scipy.fft
import numpy as np

a = np.exp(2j * np.pi * np.arange(8) / 8)
print(a)
b = scipy.fft.fft(a)
b
```

```
[ 1.00000000e+00+0.00000000e+00j  7.07106781e-01+7.07106781e-01j
  6.12323400e-17+1.00000000e+00j -7.07106781e-01+7.07106781e-01j
 -1.00000000e+00+1.22464680e-16j -7.07106781e-01-7.07106781e-01j
 -1.83697020e-16-1.00000000e+00j  7.07106781e-01-7.07106781e-01j]
```

Out[110]:
```
array([-3.44509285e-16+1.22464680e-16j,  8.00000000e+00-9.95431023e-16j,
        3.44509285e-16+1.22464680e-16j,  0.00000000e+00+1.22464680e-16j,
        9.95799250e-17+1.22464680e-16j, -8.88178420e-16+2.60642944e-16j,
       -9.95799250e-17+1.22464680e-16j,  0.00000000e+00+1.22464680e-16j])
```

In [111]:
```
scipy.fft.ifft(b)
```

Out[111]:
```
array([ 1.00000000e+00+1.23259516e-32j,  7.07106781e-01+7.07106781e-01j,
        6.12323400e-17+1.00000000e+00j, -7.07106781e-01+7.07106781e-01j,
       -1.00000000e+00+1.22464680e-16j, -7.07106781e-01-7.07106781e-01j,
       -1.83697020e-16-1.00000000e+00j,  7.07106781e-01-7.07106781e-01j])
```

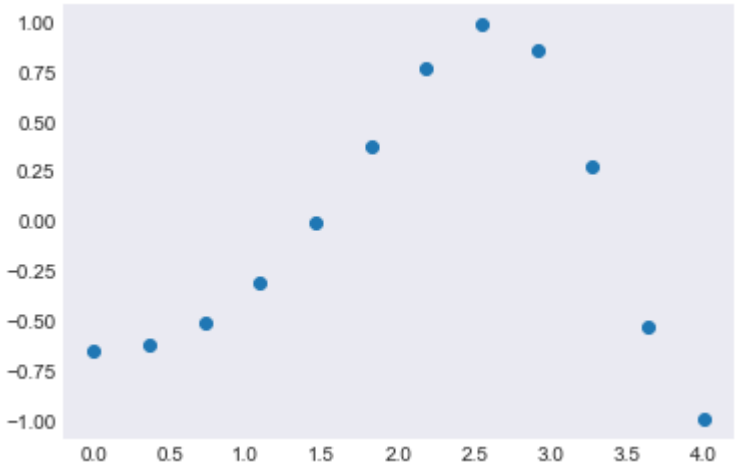
3. Interpolation (scipy.interpolate)

Sub-package for objects used in interpolation.

As listed below, this sub-package contains spline functions and classes, 1-D and multidimensional (univariate and multivariate) interpolation classes, Lagrange and Taylor polynomial interpolators, and wrappers for FITPACK and DFITPACK functions.

- interp1d(x, y[, kind, axis, copy, …]): Interpolate a 1-D function.
- UnivariateSpline(x, y[, w, bbox, k, s, ext, …]): 1-D smoothing spline fit to a given set of data points.

In [112]:
```
from scipy import interpolate as intp
import matplotlib.pyplot as plt
x = np.linspace(0, 4, 12)
y = np.cos(x**2/3 + 4)
plt.plot(x, y, 'o')
plt.show()
```



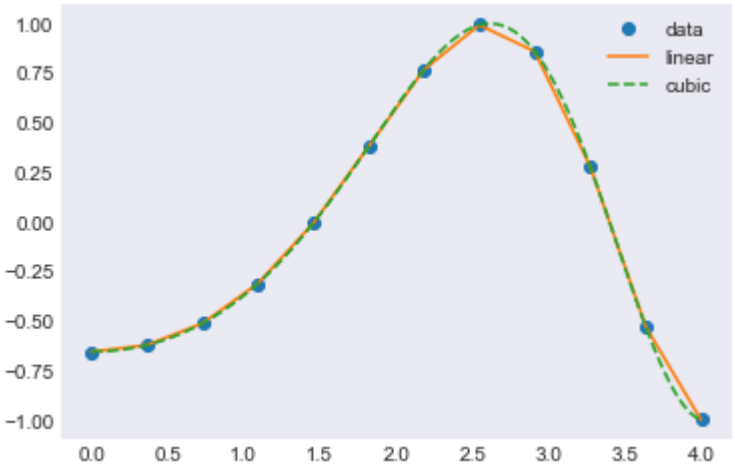
3.1 scipy.interpolate.interp1d

scipy.interpolate.interp1d(x, y, kind='linear', axis=- 1, copy=True, bounds_error=None, fill_value=nan, assume_sorted=False)

Interpolate a 1-D function.

- x and y are arrays of values used to approximate some function f: y = f(x). This class returns a function whose call method uses interpolation to find the value of new points.
- kind: Specifies the kind of interpolation as a string (‘linear’, ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘previous’, ‘next’, where ‘zero’, ‘slinear’, ‘quadratic’ and ‘cubic’ refer to a spline interpolation of zeroth, first, second or third order; ‘previous’ and ‘next’ simply return the previous or next value of the point) or as an integer specifying the order of the spline interpolator to use. Default is ‘linear’.

In [113]:
```
f1 = intp.interp1d(x, y, kind = 'linear')
f2 = intp.interp1d(x, y, kind = 'cubic')
xnew = np.linspace(0, 4, 300)
plt.plot(x, y, 'o', xnew, f1(xnew), '-', xnew, f2(xnew), '--')
plt.legend(['data', 'linear', 'cubic','nearest'], loc = 'best')
plt.show()
```



3.2 scipy.interpolate.UnivariateSpline

scipy.interpolate.UnivariateSpline(x, y, w=None, bbox=[None, None], k=3, s=None, ext=0, check_finite=False)

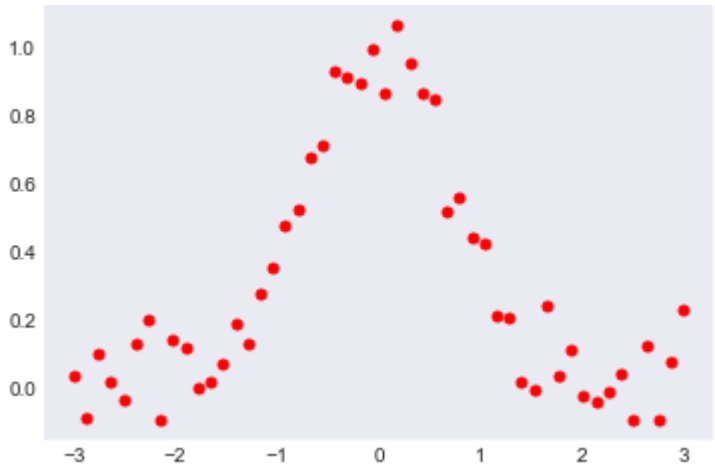
1-D smoothing spline fit to a given set of data points.

Methods:

- set_smoothing_factor(self, s): Continue spline computation with the given smoothing factor s and with the knots found at the last call.This routine modifies the spline in place.

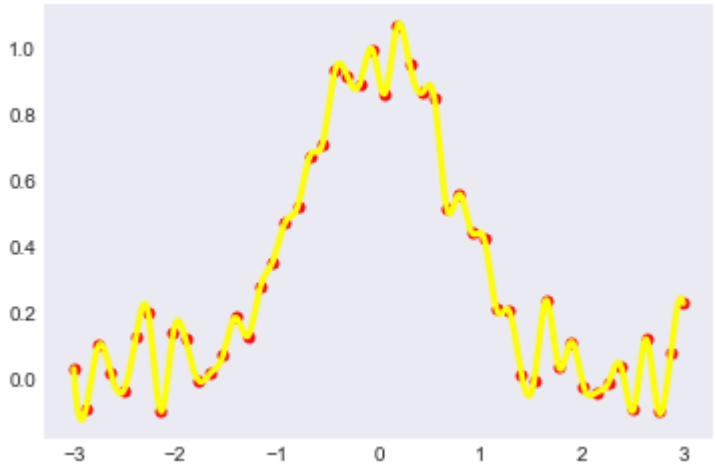
In [114]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import UnivariateSpline
x = np.linspace(-3, 3, 50)
y = np.exp(-x**2) + 0.1 * np.random.randn(50) # add noise by random method
plt.plot(x, y, 'ro', ms=5)
```

Out[114]:
```
[<matplotlib.lines.Line2D at 0x2a02b3ec308>]
```

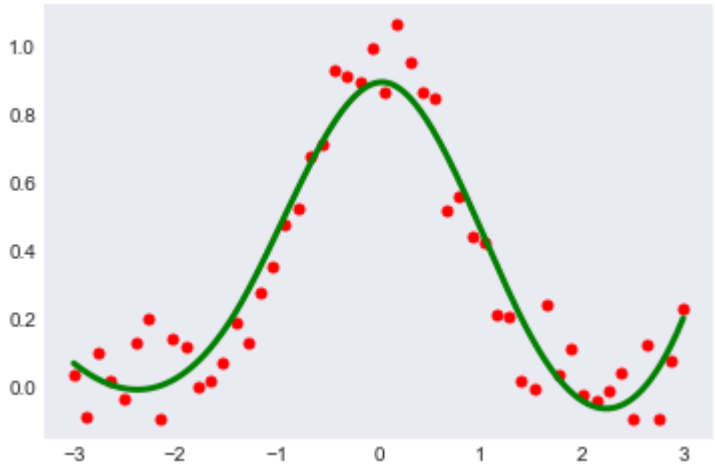


In [115]:
```
plt.plot(x, y, 'ro', ms=5)

xs = np.linspace(-3, 3, 1000)
spl = UnivariateSpline(x, y)
# set_smoothing_factor as 0
spl.set_smoothing_factor(0)
plt.plot(xs, spl(xs), 'yellow', lw=3) # yellow line
```

Out[115]:
```
[<matplotlib.lines.Line2D at 0x2a02b4257c8>]
```

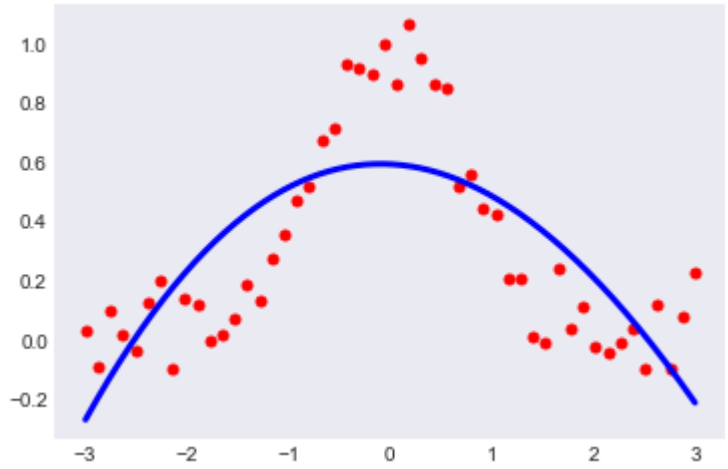


In [116]:
```
plt.plot(x, y, 'ro', ms=5)

# set_smoothing_factor as 0.5
spl.set_smoothing_factor(0.5)
plt.plot(xs, spl(xs), 'green', lw=3) # green line


plt.show()
```



In [117]:
```
plt.plot(x, y, 'ro', ms=5)

# set_smoothing_factor as 100
spl.set_smoothing_factor(100)
plt.plot(xs, spl(xs), 'blue', lw=3) # blue line
```

Out[117]:
```
[<matplotlib.lines.Line2D at 0x2a0282d9e08>]
```



4. Linear Algebra (scipy.linalg)

scipy.linalg contains all the functions in numpy.linalg. plus some other more advanced ones not contained in numpy.linalg.

Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for numpy this is optional. Therefore, the scipy version might be faster depending on how numpy was installed.

Therefore, unless you don't want to add scipy as a dependency to your numpy program, use scipy.linalg instead of numpy.linalg.

4.1 Solving a linear system

Solving linear systems of equations is straightforward using the scipy command linalg.solve. This command expects an input matrix and a right-hand side vector. The solution vector is then computed. An option for entering a symmetric matrix is offered, which can speed up the processing when applicable.

In [118]:
```
from scipy import linalg

a = np.array([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
b = np.array([10, 8, 3])
print(a)
print(b)
```

```
[[1 3 5]
 [2 5 1]
 [2 3 8]]
[10  8  3]
```

$$
\begin{aligned}
x + 3y + 5z &= 10 \\
2x + 5y + z &= 8 \\
2x + 3y + 8z &= 3
\end{aligned}
$$

$$
\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}
$$

In [119]:
```
x = linalg.solve(a, b)
x
```

Out[119]:
```
array([-9.28,  5.16,  0.76])
```

4.2 Finding the determinant

The determinant of a square matrix A is often denoted |A| and is a quantity often used in linear algebra.

In [120]:
```
A = np.array([[1,2],[3,4]])
A
```

Out[120]:
```
array([[1, 2],
       [3, 4]])
```

In [121]:
```
linalg.det(A)
```

Out[121]:
```
-2.0
```

5. Decompositions

5.1 Eigenvalues and eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. In one popular form, the eigenvalue-eigenvector problem is to find for some square matrix A, scalars λ and corresponding vectors v, such that

$$Av = \lambda v$$

In [122]:
```
A = np.array([[3,4],[7,8]])

eigenvalue, eigenvector = linalg.eig(A)

print(eigenvalue)
print(eigenvector)
```

```
[-0.35234996+0.j 11.35234996+0.j]
[[-0.76642628 -0.43192981]
 [ 0.64233228 -0.90190722]]
```

5.2 Singular value decomposition

Singular value decomposition (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square. Let A be an $M \times N$ matrix with M and N arbitrary. The matrices $A^H A$ and AA^H are square hermitian matrices of size $N \times N$ and $M \times M$, respectively. It is known that the eigenvalues of square hermitian matrices are real and non-negative. In addition, there are at most $min(M, N)$ identical non-zero eigenvalues of $A^H A$ and AA^H. Define these positive eigenvalues as σ_i^2. The square-root of these are called singular values of A. The eigenvectors of $A^H A$ are collected by columns into an $N \times N$ unitary matrix V, while the eigenvectors of AA^H are collected by columns in the unitary matrix U, the singular values are collected in an $M \times N$ zero matrix Σ with main diagonal entries set to the singular values. Then

$$A = U\Sigma V^H$$

is the singular value decomposition of A. Every matrix has a singular value decomposition. Sometimes, the singular values are called the spectrum of A. The command linalg.svd will return U, V^H, and σ_i as an array of the singular values. To obtain the matrix Σ, use linalg.diagsvd. The

following example illustrates the use of linalg.svd:

In [123]:
```
A = np.array([[1,2,3],[4,5,6]])
A
```

Out[123]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [124]:
```
M,N = A.shape
U,s,Vh = linalg.svd(A)
Sig = linalg.diagsvd(s,M,N)
U
```

Out[124]:
```
array([[-0.3863177 , -0.92236578],
       [-0.92236578,  0.3863177 ]])
```

In [125]:
```
Sig
```

Out[125]:
```
array([[9.508032  , 0.        , 0.        ],
       [0.        , 0.77286964, 0.        ]])
```

In [126]:
```
Vh
```

Out[126]:
```
array([[-0.42866713, -0.56630692, -0.7039467 ],
       [ 0.80596391,  0.11238241, -0.58119908],
       [ 0.40824829, -0.81649658,  0.40824829]])
```

In [127]:
```
U.dot(Sig.dot(Vh)) #check computation
```

Out[127]:
```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

In [128]:
```
U@Sig@Vh
```

Out[128]:
```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

05 scikit-learn

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

1. Fitting and predicting: estimator basics

Scikit-learn provides dozens of built-in machine learning algorithms and models, called estimators. Each estimator can be fitted to some data using its fit method.

1.1 LinearRegression

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the features. In mathematical notation, if $\hat{y}$ is the predicted value.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \ldots + w_p x_p$$

Across the module, we designate the vector $w_0 = (w_1, \ldots, w_p)$ as coef_ and w_0 as intercept_.

LinearRegression fits a linear model with coefficients to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$min||Xw - y||_2^2$$

LinearRegression will take in its fit method arrays X, y and will store the coefficients of the linear model in its coef_ member.

In [129]:
```
from sklearn import linear_model

X = [[0, 0], [1, 1], [2, 2]]
y = [0, 1, 2]

reg = linear_model.LinearRegression()
reg.fit(X, y)

reg.coef_
```

Out[129]:
```
array([0.5, 0.5])
```

2. Transformers and pre-processors

In scikit-learn, pre-processors and transformers follow the same API as the estimator objects (they actually all inherit from the same BaseEstimator class). The transformer objects don't have a predict method but rather a transform method that outputs a newly transformed sample matrix X:

In [130]:
```
from sklearn.preprocessing import StandardScaler
X = [[0, 0], [1, 1], [2, 2]]

StandardScaler().fit(X).transform(X)
```

Out[130]:
```
array([[-1.22474487, -1.22474487],
       [ 0.        ,  0.        ],
       [ 1.22474487,  1.22474487]])
```

In [131]:
```
from sklearn.preprocessing import MinMaxScaler

data = MinMaxScaler(feature_range=(0., 1.)).fit_transform(X)
data
```

Out[131]:
```
array([[0. , 0. ],
       [0.5, 0.5],
       [1. , 1. ]])
```

3. Pipelines: chaining pre-processors and estimators

Transformers and estimators (predictors) can be combined together into a single unifying object: a Pipeline. The pipeline offers the same API as a regular estimator: it can be fitted and used for prediction with fit and predict. As we will see later, using a pipeline will also prevent you from data leakage, i.e. disclosing some testing data in your training data.

In [132]:
```
from sklearn.pipeline import make_pipeline

# create a pipeline object
pipe = make_pipeline(
    StandardScaler(),
    linear_model.LinearRegression()
)

# fit the whole pipeline
pipe.fit(X, y)

# coefficient
pipe[1].coef_
```

Out[132]:
```
array([0.40824829, 0.40824829])
```

4. Model evaluation

Fitting a model to some data does not entail that it will predict well on unseen data. This needs to be directly evaluated.

In [133]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
pipe.fit(x_train, y_train)

mean_squared_error(y_test, pipe.predict(x_test))
```

Out[133]:
```
4.930380657631324e-32
```

In [ ]:

In [ ]:

In [ ]: