**NumPy Numerical Python**

**Python Array**

>>> import array

>>> l=list(range(10))

>>> a=array.array('i',l)

>>> a

array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Python array object provide efficient storage of array based data.

Numpy add to this efficient **operation** on data

**Import numpy as np**

**np.array([1,2,3,4,5])**

Where as in numpy is constrained to arrays that all contain the same type.

If type do not match , Numpy will upcast if possible

For example considered the following scripts

**>>> x=np.array([3.14,1,2,3,4])**

>>> x

If we want explicitly set the data type use the following

**>>> x=np.array([1,2,3,4,5],dtype=float)**

**>>> x**

Unlike python list, Numpy arrays can explicitly be multidimensional

>>> x=np.array([range(i,i+3)for i in [2,3,4]])

>>> x

>>> import numpy as np

**>>> x=np.ones((3,3),dtype=int)**

>>> x

array([[1, 1, 1],

[1, 1, 1],

[1, 1, 1]])

>>> print(x)

[[1 1 1]

[1 1 1]

[1 1 1]]

**>>> x=np.zeros((3,4),dtype=int) # create 3,4 array filled up with 0s**

>>> print(x)

[[0 0 0 0]

[0 0 0 0]

[0 0 0 0]]

**>>> x=np.full((3,3),2) # create 3,3 array filled up with 2**

>>> x

array([[2, 2, 2],

[2, 2, 2],

[2, 2, 2]])

>>>

Create an array of five values evenly spaced between two numbers

**>>> np.linspace(0,1,5)**

array([0. , 0.25, 0.5 , 0.75, 1. ])

**>>> np.linspace(0,10,5)**

array([ 0. , 2.5, 5. , 7.5, 10. ])

**>>> np.linspace(0,1,5)**

array([0. , 0.25, 0.5 , 0.75, 1. ])

>>>

**>>> np.random.random((3,4))**

array([[0.95437831, 0.38693225, 0.34366396, 0.49490602],

[0.30876109, 0.36179099, 0.23686035, 0.54103793],

[0.87017639, 0.24068009, 0.87335365, 0.3318868 ]])

**Create a 3,3 array of uniformly distributed random values.**

**>>> np.random.random((3,3))**

array([[0.89303946, 0.87538087, 0.7743825 ],

[0.74368853, 0.89526505, 0.8121449 ],

[0.76855698, 0.96510605, 0.15807987]])

Create 3,3 array of normally distributed random values.

**>>> np.random.normal(5,2.6(3,3)) where 5 is mean and 2.6 is sd**

array([[ 0.22497204, -1.4127627 , -0.9921821 ],

[ 1.57896044, 0.94077256, 0.63433971],

[ 0.73959588, 0.5638651 , 0.13927189]])

>>>

**>>> x=np.random.randint(0,10,(3,3))**

>>> x

array([[1, 2, 3],

[0, 6, 7],

[2, 7, 5]])

>>>

**To create identical matrix**

**>>> x=np.eye(3,3)**

>>> x

array([[1., 0., 0.],

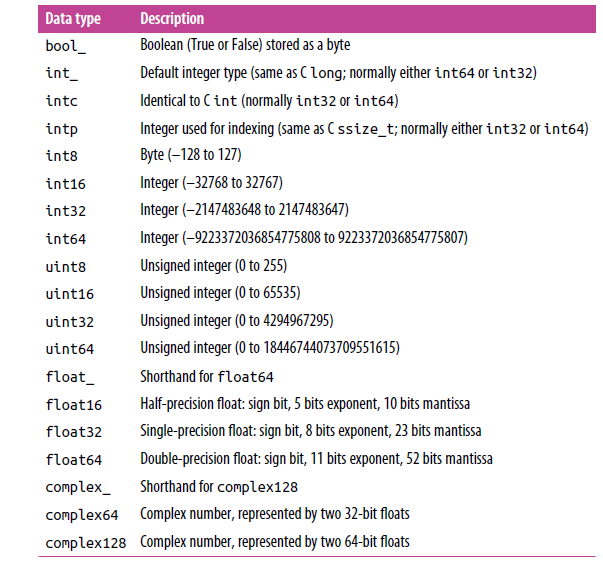
[0., 1., 0.],

[0., 0., 1.]])

>>>

**Standard Data types –NumPy**

NumPy array contains value of a single type.



**Array Attributes**

**Shape**

**>>> x=np.random.normal(3,2,(3,4))**

>>> x

array([[6.18912107, 3.46086834, 2.87017931, 1.0620395 ],

[4.18248561, 1.43444899, 2.11153435, 2.30962768],

[1.2363989 , 2.11469352, 1.9181674 , 0.35354526]])

>>> x.shape

(3, 4)

**>>> x.reshape(4,3)**

array([[6.18912107, 3.46086834, 2.87017931],

[1.0620395 , 4.18248561, 1.43444899],

[2.11153435, 2.30962768, 1.2363989 ],

[2.11469352, 1.9181674 , 0.35354526]])

**>>> x.shape=(4,3)**

>>> x

array([[6.18912107, 3.46086834, 2.87017931],

[1.0620395 , 4.18248561, 1.43444899],

[2.11153435, 2.30962768, 1.2363989 ],

[2.11469352, 1.9181674 , 0.35354526]])

>>>

>>> b=x.T

>>>b

Transpose makes arrays non contagious

b.view()

>>> import numpy as np

>>> x=np.random.normal(1,2,(3,4))

>>> x

array([[ 1.22991834, 1.69062528, -2.46991753, 4.31670221],

[ 5.59954305, 0.05772948, 3.52543097, -1.3410309 ],

[ 3.1315784 , -0.39987477, 1.28815823, 1.79708421]])

>>> b=x.T

>>> b

array([[ 1.22991834, 5.59954305, 3.1315784 ],

[ 1.69062528, 0.05772948, -0.39987477],

[-2.46991753, 3.52543097, 1.28815823],

[ 4.31670221, -1.3410309 , 1.79708421]])

**>>> b.shape=(3,4)**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: incompatible shape for a non-contiguous array

**>>> b.view()**

array([[ 1.22991834, 5.59954305, 3.1315784 ],

[ 1.69062528, 0.05772948, -0.39987477],

[-2.46991753, 3.52543097, 1.28815823],

[ 4.31670221, -1.3410309 , 1.79708421]])

**>>> b.reshape(3,4)**

array([[ 1.22991834, 5.59954305, 3.1315784 , 1.69062528],

[ 0.05772948, -0.39987477, -2.46991753, 3.52543097],

[ 1.28815823, 4.31670221, -1.3410309 , 1.79708421]])

>>> **b.shape=(3,4)**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: incompatible shape for a non-contiguous array

>>>

**numpy.ravel**

Return a contiguous flattened array. A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

**>>> x=np.array([[1,2,3,4],[5,6,7,8]])**

>>> x

array([[1, 2, 3, 4],

[5, 6, 7, 8]])

**>>> r=np.ravel((x))**

>>> print(r)

[1 2 3 4 5 6 7 8]

>>>

**NumPy Array Attributes**

**Each array has attributes**

**ndim the number of dimension**

**Size the size of each dimension**

**Shape the total size of the array**

**Considered the following example**

**>>> x1=np.random.randint(10,size=5)**

>>> x1

array([7, 3, 4, 5, 8])

**>>> x2=np.random.randint(10,size=(3,4))**

>>> x2

array([[6, 6, 4, 9],

[1, 4, 9, 4],

[2, 8, 3, 4]])

**>>> x3=np.random.randint(10,size=(2,3,4))**

>>> x3

array([[[7, 1, 0, 3],

[4, 3, 8, 9],

[1, 6, 0, 4]],

[[9, 8, 9, 0],

[2, 9, 7, 0],

[9, 7, 8, 0]]])

**>>> x3.shape**

(2, 3, 4)

**>>> x3.size**

24

**>>> x3.ndim**

3

>>>

**Additional Attributes are**

**dtype** the datatype of the array

**itemsize** the size of each array element(bytes)

**nbytes** the total size of the array 🡺 itemsize \* size

**>>> print(x3.dtype)**

int32

**>>> x3.itemsize**

4

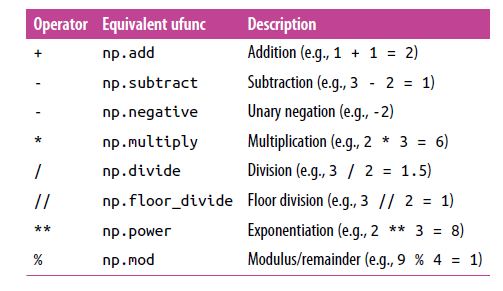
**>>> x3.nbytes**

96

>>>

**NumPy Arithmetic Operations**

Input arrays for performing arithmetic operations such as add(), subtract(), multiply(), and divide() must be either of the same shape or should conform to array broadcasting rules.



>>> import numpy as np

>>> # Arithmetic Operations

...

>>> x=np.random.randint(9,size=(3,3))

>>> x

array([[7, 7, 6],

[2, 3, 4],

[3, 4, 6]])

>>> y=np.random.randint(9,size=(3,3))

>>> y

array([[3, 2, 6],

[0, 0, 4],

[2, 7, 2]])

>>> # Addition

...

>>> np.add(x,y)

array([[10, 9, 12],

[ 2, 3, 8],

[ 5, 11, 8]])

>>> # subtraction

>>> np.subtract(x,y)

array([[ 4, 5, 0],

[ 2, 3, 0],

[ 1, -3, 4]])

>>> np.multiply(x,y)

array([[21, 14, 36],

[ 0, 0, 16],

[ 6, 28, 12]])

>>> np.divide(x,y)

\_\_main\_\_:1: RuntimeWarning: divide by zero encountered in true\_divide

array([[2.33333333, 3.5 , 1. ],

[ inf, inf, 1. ],

[1.5 , 0.57142857, 3. ]])

>>> np.floor\_divide(x,y)

\_\_main\_\_:1: RuntimeWarning: divide by zero encountered in floor\_divide

array([[2, 3, 1],

[0, 0, 1],

[1, 0, 3]], dtype=int32)

>>> x

array([[7, 7, 6],

[2, 3, 4],

[3, 4, 6]])

>>> y

array([[3, 2, 6],

[0, 0, 4],

[2, 7, 2]])

>>> y[1][0]=2

>>> y

array([[3, 2, 6],

[2, 0, 4],

[2, 7, 2]])

>>> y[1][1]=3

>>> y

array([[3, 2, 6],

[2, 3, 4],

[2, 7, 2]])

>>> np.floor\_divide(x,y)

array([[2, 3, 1],

[1, 1, 1],

[1, 0, 3]], dtype=int32)

>>> np.divide(x,y)

array([[2.33333333, 3.5 , 1. ],

[1. , 1. , 1. ],

[1.5 , 0.57142857, 3. ]])

**reciprocal**

**This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 1, the result is always 0**

>>> np.reciprocal(x)

array([[0, 0, 0],

[0, 0, 0],

[0, 0, 0]], dtype=int32)

>>>

**power()**

This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array.

>>> x

array([[7, 7, 6],

[2, 3, 4],

[3, 4, 6]])

**>>> np.power(x,2)**

array([[49, 49, 36],

[ 4, 9, 16],

[ 9, 16, 36]], dtype=int32)

>>>

**mod()**

This function returns the remainder of division of the corresponding elements in the input array. The function **numpy.remainder()** also produces the same result.

**>>> x**

**array([[7, 7, 6],**

**[2, 3, 4],**

**[3, 4, 6]])**

**>>> y**

**array([[3, 2, 6],**

**[2, 3, 4],**

**[2, 7, 2]])**

**>>> np.mod(x,y)**

**array([[1, 1, 0],**

**[0, 0, 0],**

**[1, 4, 0]], dtype=int32)**

**Numpy Understand python built in arithmetic operators, it also understand python’s built in absolute value functions.**

**>>> np.array([-1,-2,1,2,3,4,5])**

array([-1, -2, 1, 2, 3, 4, 5])

**>>> x=np.array([-1,-2,1,2,3,4,5])**

**>>> np.absolute(x)**

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

**>>> np.abs(x)**

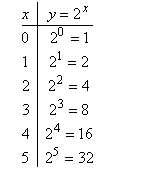
array([1, 2, 1, 2, 3, 4, 5])

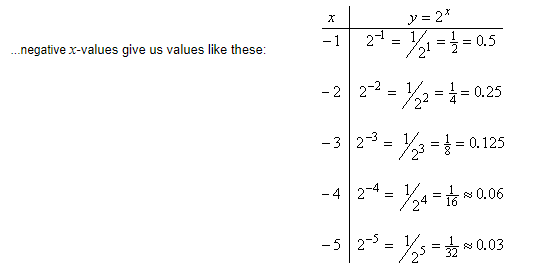
**>>>**

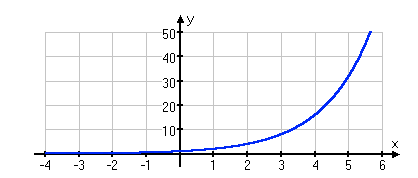
Exponential functions look somewhat similar to functions you have seen before, in that they involve exponents, but there is a big difference, in that the variable is now the power, rather than the base. Previously, you have dealt with such functions as

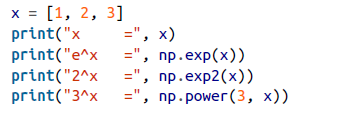
f(x) = x2, where the variable x was the base and the number 2 was the power. In the case of exponentials, however, you will be dealing with functions such as g(x) = 2x, where the base is the fixed number, and the power is the variable.

Let's look more closely at the function g(x) = 2x. To evaluate this function, we operate as usual, picking values of x.







****

**>>> x=np.random.randint(9,size=(3,3))**

**>>> x**

array([[5, 5, 1],

[6, 0, 7],

[0, 4, 7]])

**>>> np.exp(x)**

array([[1.48413159e+02, 1.48413159e+02, 2.71828183e+00],

[4.03428793e+02, 1.00000000e+00, 1.09663316e+03],

[1.00000000e+00, 5.45981500e+01, 1.09663316e+03]])

**Partition**

Creates a copy of the array with its elements rearranged in such a way that the value of the element in k-th position is in the position it would be in a sorted array. All elements smaller than the k-th element are moved before this element and all equal or greater are moved behind it.

>>> x=np.random.randint(12,size=(3,3))

>>> x

array([[ 2, 4, 2],

[ 7, 10, 6],

[ 7, 8, 0]])

**>>> y=np.partition(x,1)**

>>> y

array([[ 2, 2, 4],

[ 6, 7, 10],

[ 0, 7, 8]])

**Cross product**

**>>> x**

**array([[ 0, 8, 0],**

**[ 1, 3, 10],**

**[10, 8, 8]])**

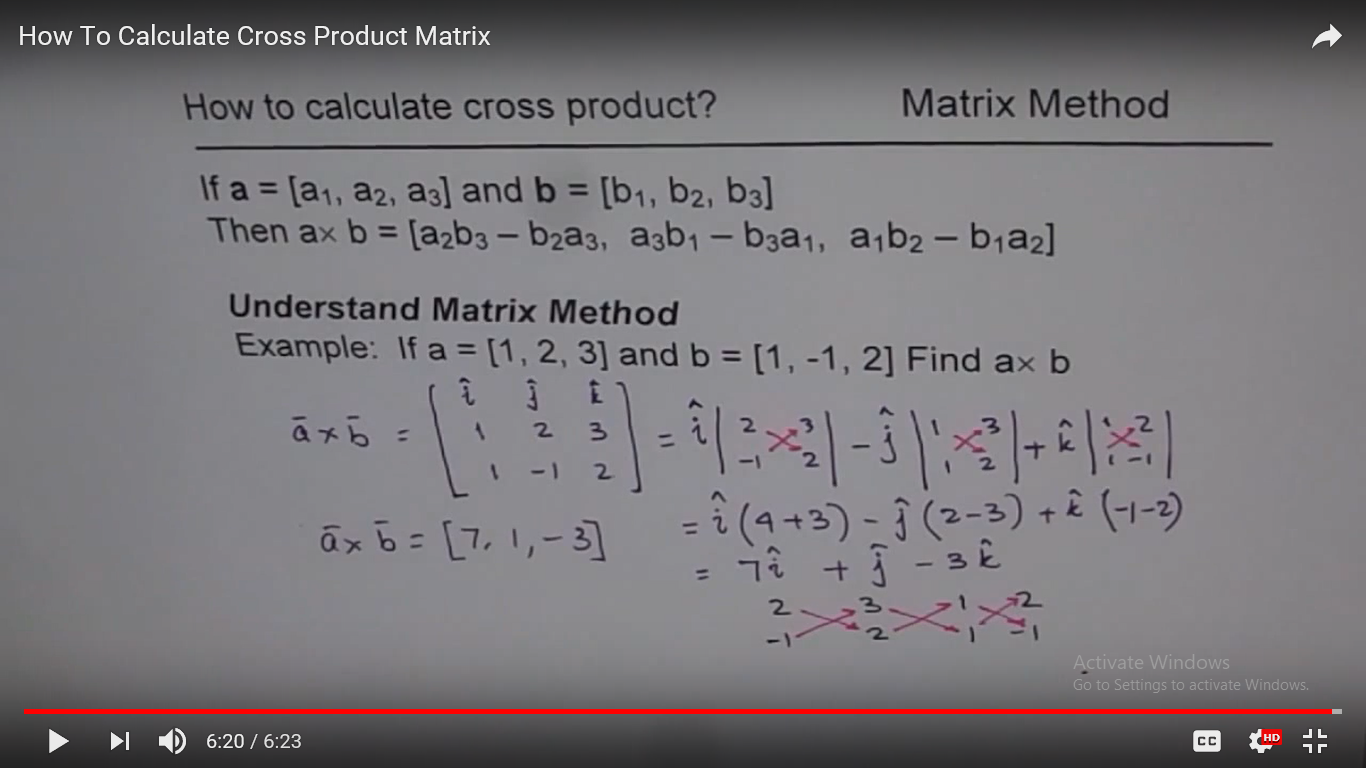
**>>> y**

**array([[10, 9, 6],**

**[10, 5, 10],**

**[10, 1, 7]])**

**Cross Product** https://www.youtube.com/watch?v=4LKZw2\_SUpA

****

**Array Indexing - Accessing Single Elements from the array**

We can access the ith value by specifying the desired index in the square bracket, just as the python list.

**>>> x1**

array([7, 3, 4, 5, 8])

**>>> x1[0]**

7

**>>> x1[1]**

3

**>>> x1[2]**

4

**>>> x1[3]**

5

**>>> x1[4]**

8

**>>> x1[-1]**

8

**>>> x2[1][2]**

9

**>>> x2**

array([[6, 6, 4, 9],

[1, 4, 9, 4],

[2, 8, 3, 4]])

**>>> x2[0][1]**

6

**>>> x2[0][0]**

6

**>>> x2[0][2]**

4

**>>> x2[0][3]**

9

**>>> x2[1][3]**

4

You can also modify values using any of the array index notation

>>> x2[0][0]=7

>>> x2

Unlike python list, numpy array has fixed type, this mean if we try to assign floating point values to the integer array the value will automcatically truncated

X2[0][0]=2.31 # this will be truncated

**Array Slicing**

Numpy slicing syntax follows the python list.

To access the slice of an array x use the following syntax

**x[start:stop:step]**

**>>> x=np.arange(10)**

**>>> x**

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

**>>> x[0:9]**

array([0, 1, 2, 3, 4, 5, 6, 7, 8])

**>>> x[0:5]**

array([0, 1, 2, 3, 4])

**>>> x[0:9:2]**

array([0, 2, 4, 6, 8])

**>>> x[1::2]**

array([1, 3, 5, 7, 9])

**>>> x[::-1]**

array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

**>>>**

**Transpose**

This function permutes the dimension of the given array. It returns a view wherever possible. The function takes the following parameters.

**>>> import numpy as np**

**>>> x=np.arange(12).reshape(3,4)**

**>>> x**

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

**>>> np.transpose(x)**

array([[ 0, 4, 8],

[ 1, 5, 9],

[ 2, 6, 10],

[ 3, 7, 11]])

**>>>**

**String Operations**

|  |  |
| --- | --- |
| [**add**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.add.html#numpy.core.defchararray.add)(x1, x2) | Return element-wise string concatenation for two arrays of str or unicode. |
| [**multiply**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.multiply.html#numpy.core.defchararray.multiply)(a, i) | Return (a \* i), that is string multiple concatenation, element-wise. |
|  |  |
| [**capitalize**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.capitalize.html#numpy.core.defchararray.capitalize)(a) | Return a copy of *a* with only the first character of each element capitalized. |
| [**center**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.center.html#numpy.core.defchararray.center)(a, width[, fillchar]) | Return a copy of *a* with its elements centered in a string of length *width*. |
| [**decode**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.decode.html#numpy.core.defchararray.decode)(a[, encoding, errors]) | Calls *str.decode* element-wise. |
| [**encode**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.encode.html#numpy.core.defchararray.encode)(a[, encoding, errors]) | Calls *str.encode* element-wise. |
| [**join**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.join.html#numpy.core.defchararray.join)(sep, seq) | Return a string which is the concatenation of the strings in the sequence *seq*. |
| [**ljust**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.ljust.html#numpy.core.defchararray.ljust)(a, width[, fillchar]) | Return an array with the elements of *a* left-justified in a string of length *width*. |
| [**lower**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.lower.html#numpy.core.defchararray.lower)(a) | Return an array with the elements converted to lowercase. |
| [**lstrip**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.lstrip.html#numpy.core.defchararray.lstrip)(a[, chars]) | For each element in *a*, return a copy with the leading characters removed. |
| [**partition**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.partition.html#numpy.core.defchararray.partition)(a, sep) | Partition each element in *a* around *sep*. |
| [**replace**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.replace.html#numpy.core.defchararray.replace)(a, old, new[, count]) | For each element in *a*, return a copy of the string with all occurrences of substring *old* replaced by **new**. |
| [**rjust**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.rjust.html#numpy.core.defchararray.rjust)(a, width[, fillchar]) | Return an array with the elements of *a* right-justified in a string of length *width*. |
|  |  |
| [**rsplit**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.rsplit.html#numpy.core.defchararray.rsplit)(a[, sep, maxsplit]) | For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string. |
| [**rstrip**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.rstrip.html#numpy.core.defchararray.rstrip)(a[, chars]) | For each element in *a*, return a copy with the trailing characters removed. |
| [**split**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.split.html#numpy.core.defchararray.split)(a[, sep, maxsplit]) | For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string. |
| [**splitlines**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.splitlines.html#numpy.core.defchararray.splitlines)(a[, keepends]) | For each element in *a*, return a list of the lines in the element, breaking at line boundaries. |
| [**strip**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.strip.html#numpy.core.defchararray.strip)(a[, chars]) | For each element in *a*, return a copy with the leading and trailing characters removed. |
| [**swapcase**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.swapcase.html#numpy.core.defchararray.swapcase)(a) | Return element-wise a copy of the string with uppercase characters converted to lowercase and vice versa. |
| [**title**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.title.html#numpy.core.defchararray.title)(a) | Return element-wise title cased version of string or unicode. |
| [**translate**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.translate.html#numpy.core.defchararray.translate)(a, table[, deletechars]) | For each element in *a*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table. |
| [**upper**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.upper.html#numpy.core.defchararray.upper)(a) | Return an array with the elements converted to uppercase. |
| [**zfill**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.zfill.html#numpy.core.defchararray.zfill)(a, width) | Return the numeric string left-filled with zeros |

**Comparison**

Unlike the standard numpy comparison operators, the ones in the *char* module strip trailing whitespace characters before performing the comparison.

|  |  |
| --- | --- |
| [**equal**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.equal.html#numpy.core.defchararray.equal)(x1, x2) | Return (x1 == x2) element-wise. |
| [**not\_equal**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.not_equal.html#numpy.core.defchararray.not_equal)(x1, x2) | Return (x1 != x2) element-wise. |
| [**greater\_equal**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.greater_equal.html#numpy.core.defchararray.greater_equal)(x1, x2) | Return (x1 >= x2) element-wise. |
| [**less\_equal**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.less_equal.html#numpy.core.defchararray.less_equal)(x1, x2) | Return (x1 <= x2) element-wise. |
| [**greater**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.greater.html#numpy.core.defchararray.greater)(x1, x2) | Return (x1 > x2) element-wise. |
| [**less**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.less.html#numpy.core.defchararray.less)(x1, x2) | Return (x1 < x2) element-wise. |
|  |  |

**String information**

|  |  |
| --- | --- |
| [**count**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.count.html#numpy.core.defchararray.count)(a, sub[, start, end]) | Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. |
| [**find**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.find.html#numpy.core.defchararray.find)(a, sub[, start, end]) | For each element, return the lowest index in the string where substring *sub* is found. |
| [**index**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.index.html#numpy.core.defchararray.index)(a, sub[, start, end]) | Like [**find**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.find.html#numpy.core.defchararray.find), but raises *ValueError* when the substring is not found. |
| [**isalpha**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.isalpha.html#numpy.core.defchararray.isalpha)(a) | Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise. |
| [**isdecimal**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.isdecimal.html#numpy.core.defchararray.isdecimal)(a) | For each element, return True if there are only decimal characters in the element. |
| [**isdigit**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.isdigit.html#numpy.core.defchararray.isdigit)(a) | Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise. |
| [**islower**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.islower.html#numpy.core.defchararray.islower)(a) | Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise. |
| [**isnumeric**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.isnumeric.html#numpy.core.defchararray.isnumeric)(a) | For each element, return True if there are only numeric characters in the element. |
| [**isspace**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.isspace.html#numpy.core.defchararray.isspace)(a) | Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise. |
| [**istitle**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.istitle.html#numpy.core.defchararray.istitle)(a) | Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise. |
| [**isupper**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.isupper.html#numpy.core.defchararray.isupper)(a) | Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise. |
| [**rfind**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.rfind.html#numpy.core.defchararray.rfind)(a, sub[, start, end]) | For each element in *a*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*]. |
| [**rindex**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.rindex.html#numpy.core.defchararray.rindex)(a, sub[, start, end]) | Like [**rfind**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.rfind.html#numpy.core.defchararray.rfind), but raises *ValueError* when the substring *sub* is not found. |
| [**startswith**](https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.core.defchararray.startswith.html#numpy.core.defchararray.startswith)(a, prefix[, start, end]) | Returns a boolean array which is *True* where the string element in *a* starts with *prefix*, otherwise *False*. |

**c=np.array(['Asss','asd','rere'])**

c

Out[116]: array(['Asss', 'asd', 'rere'], dtype='<U4')

**np.char.count(c,'S')**

Out[117]: array([0, 0, 0])

**np.char.count(c,'s')**

Out[118]: array([3, 1, 0])

**np.char.count(c,'A')**

Out[119]: array([1, 0, 0])

c

Out[120]: array(['Asss', 'asd', 'rere'], dtype='<U4')

**np.char.isupper(c)**

Out[121]: array([False, False, False])

**np.char.isnumeric(c)**

Out[122]: array([False, False, False])

**np.char.istitle(c)**

Out[123]: array([ True, False, False])

**np.char.islower(c)**

Out[124]: array([False, True, True])

**np.char.isspace(c)**

Out[125]: array([False, False, False])

**np.char.isdigit(c)**

Out[126]: array([False, False, False])

**np.char.isalpha(c)**

Out[127]: array([ True, True, True])

**np.char.find('i18n','8',1)**

Out[128]: array(2)

**np.char.find('i18n','8',1)**

Out[129]: array(2)

**np.char.find('i18n','8',2)**

Out[130]: array(2)

**np.char.find('i18n','8')**

Out[131]: array(2)

>>> x=np.array(['Dr.Balu','sathish','arun'])

>>> x

array(['Dr.Balu', 'sathish', 'arun'], dtype='<U7')

>>> np.char.count('B',x)

array([0, 0, 0])

>>> x=np.array(['Dr.Balu','sathish','arun'])

>>> x

array(['Dr.Balu', 'sathish', 'arun'], dtype='<U7')

>>> x[0]

'Dr.Balu'

>>> x

array(['Dr.Balu', 'sathish', 'arun'], dtype='<U7')

>>> np.char.count(x,'B')

array([1, 0, 0])

>>> np.char.count(x,'a')

array([1, 1, 1])

>>> np.char.isupper(x)

array([False, False, False])

>>> np.char.islower(x)

array([False, True, True])

>>> np.char.isupper(x)

array([False, False, False])

>>> x

array(['Dr.Balu', 'sathish', 'arun'], dtype='<U7')

>>> x[0]='Balu'

>>> x

array(['Balu', 'sathish', 'arun'], dtype='<U7')

>>> np.char.isupper(x)

array([False, False, False])

>>> x[0]='BALU'

>>> x

array(['BALU', 'sathish', 'arun'], dtype='<U7')

>>> np.char.isupper(x)

array([ True, False, False])

>>> x[0]='Balu'

>>> x

array(['Balu', 'sathish', 'arun'], dtype='<U7')

>>> np.char.isupper(x)

array([False, False, False])

>>> x[0]='BALU'

>>> np.char.isupper(x)

array([ True, False, False])

>>> x[1]='SATHISH'

>>> X

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'X' is not defined

>>> x

array(['BALU', 'SATHISH', 'arun'], dtype='<U7')

>>> np.char.isupper(x)

array([ True, True, False])

>>> np.char.islower(x)

array([False, False, True])

>>> np.char.isnumeric(x)

array([False, False, False])

>>> np.char.isalpha(x)

array([ True, True, True])

>>> np.char.isdecimal(x)

array([False, False, False])

>>> np.char.isdigit(x)

array([False, False, False])

>>> np.char.find('sivam','m')

array(4)

>>> np.char.find('i18n','8',1)

array(2)

>>> np.char.find('sivam','m',1)

array(4)

>>> np.char.find('smvam','m',2)

array(4)

>>> np.char.find('smvam','m')

array(1)

>>> np.char.find('smvam','m',1)

array(1)

>>> np.char.find('smvam','m',2)

array(4)

>>> np.char.find('smvam','m')

array(1)

>>> np.char.find('smvam','m',2)

array(4)

>>> np.char.index('smvam','m',2)

array(4)

>>> np.char.index('smvam','n',2)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "C:\Users\I18N\AppData\Local\Programs\Python\Python36\lib\site-packages\numpy\core\defchararray.py", line 682, in index

a, integer, 'index', [sub, start] + \_clean\_args(end))

ValueError: substring not found

>>> np.char.isspace(x)

array([False, False, False])

>>> x

array(['BALU', 'SATHISH', 'arun'], dtype='<U7')

>>> x[0]=' BALU'

>>> X

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'X' is not defined

>>> x

array([' BALU', 'SATHISH', 'arun'], dtype='<U7')

>>> np.char.isspace(x)

array([False, False, False])

>>> x[0]='B ALU'

>>> x

array(['B ALU', 'SATHISH', 'arun'], dtype='<U7')

>>> np.char.isspace(x)

array([False, False, False])

>>> x[0]=''

>>> x

array(['', 'SATHISH', 'arun'], dtype='<U7')

>>> np.char.isspace(x)

array([False, False, False])

>>> x[0]=' '

>>> np.char.isspace(x)

array([ True, False, False])

>>> np.char.istitle(x)

array([False, False, False])

>>> x[0]="Balu"

>>> x

array(['Balu', 'SATHISH', 'arun'], dtype='<U7')

>>> np.char.istitle(x)

array([ True, False, False])

>>> x[2]="ArunBalachandar"

>>> x

array(['Balu', 'SATHISH', 'ArunBal'], dtype='<U7')

>>> x[2]

'ArunBal'

>>> np.char.rfind(x)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: rfind() missing 1 required positional argument: 'sub'

>>> np.char.rfind(x,'A')

array([-1, 1, 0])

>>> np.char.rfind(x,'B')

array([ 0, -1, 4])

>>> np.char.rfind(x,'a')

array([ 1, -1, 5])

>>> np.char.rfind(x,'a')

array([ 1, -1, 5])

>>> x[1]="Sathish"

>>> x

array(['Balu', 'Sathish', 'ArunBal'], dtype='<U7')

>>> np.char.rfind(x,'a')

array([1, 1, 5])

>>> np.char.startswith(x,'S')

array([False, True, False])

>>> np.char.rfind(x,'B')

array([ 0, -1, 4])

>>> np.char.startswith(x,'B')

array([ True, False, False])

>>>

**Fancy Indexing**

Passing array of indices to access multiple array elements at once.

>>> import numpy as np

>>> x=np.random.randint(100,size=10)

>>> x

array([46, 67, 84, 88, 92, 17, 50, 33, 59, 61])

Suppose if we want to access two different elements we could do like this.

>>> x[2]

84

>>> x[6]

50

Alternatively we can pass a single list or array of indices to obtain the same results.

**>>> ind=[2,6,8]**

>>> **x[ind]**

array([84, 50, 59])

>>>

**Considered the following**

>>> ind=np.array([[3,7],[4,5]])

**>>> x[ind]**

array([[88, 33],

[92, 17]])

Fancy Indexing for multiple dimension array

>>> x=np.random.randint(12,size=(3,4))

>>> x

array([[ 8, 0, 10, 1],

[ 6, 8, 7, 3],

[ 2, 9, 10, 9]])

>>> row=np.array([0,2,2])

>>> col=np.array([1,0,2])

>>> x[row,col]

**array([ 0, 2, 10])**

>>>

Notice that the first value in the array is x[0,1] that is 0, the second is x[2,0] in the array that is **2**

**>>> x**

array([[ 8, 0, 10, 1],

[ 6, 8, 7, 3],

[ 2, 9, 10, 9]])

**>>> x[2,[1,0,2]], where 2 is the second row , subsequently followed by [1,0,2]**

**Which extract the elements at the above mentioned position.**

array([ 9, 2, 10])

**Modifying value with fancy indexing**

Fancy indexing not only used to access the elements of an array. It can also be used to modify part of an array.

Considered the following script.

>>> x=np.random.randint(100,size=10)

>>> x

array([ 1, 58, 36, 30, 25, 15, 54, 5, 70, 8])

>>> ind=np.array([2,4,8])

>>> x[ind]

array([36, 25, 70])

**>>> x[ind]+=10**

>>> x

array([ 1, 58, 46, 30, 35, 15, 54, 5, 80, 8])

>>>

**Sorting**

>>> x

array([[ 5, 45, 22, 13],

[62, 33, 43, 98],

[ 4, 35, 88, 97]])

**>>> np.sort(x)**

array([[ 5, 13, 22, 45],

[33, 43, 62, 98],

[ 4, 35, 88, 97]])

**>>> np.argsort(x)**

array([[0, 3, 2, 1],

[1, 2, 0, 3],

[0, 1, 2, 3]], dtype=int64)

>>> x

array([[ 5, 45, 22, 13],

[62, 33, 43, 98],

[ 4, 35, 88, 97]])

**>>> np.sort(x,axis=0)**

array([[ 4, 33, 22, 13],

[ 5, 35, 43, 97],

[62, 45, 88, 98]])

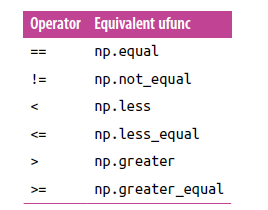
**>>> np.sort(x,axis=1)**

array([[ 5, 13, 22, 45],

[33, 43, 62, 98],

[ 4, 35, 88, 97]])

**Comparison operators as ufuncs**



**>>> x=np.arange(10)**

**>>> x**

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

**>>> x<3**

array([ True, True, True, False, False, False, False, False, False,

False])

**>>> x>3**

array([False, False, False, False, True, True, True, True, True,

True])

**>>> x>=3**

array([False, False, False, True, True, True, True, True, True,

True])

**>>> x<=3**

array([ True, True, True, True, False, False, False, False, False,

False])

**>>> x!=3**

array([ True, True, True, False, True, True, True, True, True,

True])

**>>> x==3**

array([False, False, False, True, False, False, False, False, False,

False])

We can also use the following method to access the results

**>>> np.equal(x,3)**

array([False, False, False, True, False, False, False, False, False,

False])

**>>> np.greater(x,5)**

array([False, False, False, False, False, False, True, True, True,

True])

**>>> np.less(x,2)**

array([ True, True, False, False, False, False, False, False, False,

False])

**>>> np.not\_equal(x,2)**

array([ True, True, False, True, True, True, True, True, True,

True])

**>>> np.less\_equal(x,2)**

array([ True, True, True, False, False, False, False, False, False,

False])

**>>> np.greater\_equal(x,2)**

array([False, False, True, True, True, True, True, True, True,

True])

**Computation on Arrays: Broadcasting**

**Set of rules for applying binary ufuncs(addition,subtraction,multiplication,etc.) on arrays of different size.**

**>>> a=np.array([0,1,2])**

**>>> b=np.array([5,5,5])**

**>>> a+b**

**array([5, 6, 7])**

**>>>**

**Broadcasting allows these types of binary operations to be performed on arrays of different size.**

**For example we can add just a scalar to an array**

**>>> a+5**

**array([5, 6, 7])**

**we can think this is an operation that stretches or duplicates the value 5 into the array [5,5,5] and add the results.**

**The advantage of the numpy broadcasting is that this duplication of values does not actually take place.**

**Considered the following example**

**>>> x=np.ones((3,3))**

**>>> x**

**array([[1., 1., 1.],**

**[1., 1., 1.],**

**[1., 1., 1.]])**

**>>> a=np.array([0,1,2])**

**>>> x+a**

**array([[1., 2., 3.],**

**[1., 2., 3.],**

**[1., 2., 3.]])**

Here the one dimensional array **a** is stretched or broadcast ,across the second dimension inorder to match the shape of x.

**Trigonometry functions**

NumPy provides a large number of useful ufuncs, and some of the most useful for the

data scientist are the trigonometric functions. We’ll start by defining an array of

angles:

In[15]: theta = np.linspace(0, np.pi, 3)

Now we can compute some trigonometric functions on these values:

In[16]: print("theta = ", theta)

print("sin(theta) = ", np.sin(theta))

print("cos(theta) = ", np.cos(theta))

print("tan(theta) = ", np.tan(theta))

theta = [ 0. 1.57079633 3.14159265]

sin(theta) = [ 0.00000000e+00 1.00000000e+00 1.22464680e-16]

cos(theta) = [ 1.00000000e+00 6.12323400e-17 -1.00000000e+00]

tan(theta) = [ 0.00000000e+00 1.63312394e+16 -1.22464680e-16]

The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

**>>> x=np.random.randint(12,size=(3,3))**

**>>> x**

**array([[ 7, 3, 11],**

**[ 0, 8, 7],**

**[ 9, 7, 5]])**

**>>> np.count\_nonzero(x)**

**8**

**>>> np.count\_nonzero(x<3)**

**1**

**>>> np.count\_nonzero(x>13)**

**0**

**>>> np.count\_nonzero(x)**

**8**

**>>> np.count\_nonzero(x<1)**

**1**

**If we’re interested in quickly checking whether any or all the values are true, we can use (you guessed it)**

**np.any() or np.all():**

**>>> np.any(x)**

**True**

**>>> np.any(x>1)**

**True**

**>>> x**

**array([[ 7, 3, 11],**

**[ 0, 8, 7],**

**[ 9, 7, 5]])**

**>>> #Are there any value greater than 8**

**...**

**>>> np.any(x>8)**

**True**

**>>> # Are there any value less than zero**

**...**

**>>> np.any(x<0)**

**False**

**>>> # Are there any value less than ten**

**...**

**>>> np.any(x<10)**

**True**

**>>> # Are there any value equals to five**

**...**

**>>> np.any(x==5)**

**True**

**>>>**

**>>>**

**are all values in each row greater than 8?**

**>>> np.all(x>4,axis=1)**

**array([False, False, True])**

**>>>**

**Partial Sorts: Partitioning**

Sometimes we’re not interested in sorting the entire array, but simply want to find the *K* smallest values in the array. NumPy provides this in the **np.partition** function. **np.partition** takes an array and a number *K*; the result is a new array with the smallest *K* values to the left of the partition, and the remaining values to the right, in arbitrary order:

>>> x=np.random.randint(12,size=(1,10))

>>> x

array([[ 7, 11, 0, 10, 9, 1, 3, 6, 2, 4]])

>>> np.partition(x,4)

array([[ **0, 1, 2, 3**, 4, 6, 7, 11, 10, 9]])

>>>

Note that the first four values in the resulting array are the four smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

In the previous sessions, we dove into detail on NumPy and its ndarray object, which provides efficient storage and manipulation of dense typed arrays in Python.

Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame.

DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data.

Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

**NumPy’s ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (attaching labels to data, working with missing data, etc.)**

**Pandas, and in particular its Series and DataFrame objects, builds on the NumPy array structure and provides efficient access to these sorts of “data munging” tasks that occupy much of a data scientist’s time.**

**Data wrangling, sometimes referred to as data munging, is the process of transforming and mapping data from one "raw" data form into another format with the intent of making it more appropriate and valuable for a variety of downstream purposes such as analytics. A data wrangler is a person who performs these transformation operations.**

**Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.**

**DataFrame from Series of array**

**Data Frame from List**

**DataFrame from List of Dict**

**DataFrame from Dict of List**

**DataFrame from Dict of Series**

**The Pandas Series Object**

A Pandas Series is a one-dimensional array of indexed data. It can be created from a

list or array as follows:

**>>> import numpy as np**

**>>> import pandas as pd**

**>>> data=pd.Series([0.25,0.5,0.75,1.0,1.25])**

**>>> data**

0 0.25

1 0.50

2 0.75

3 1.00

4 1.25

**dtype: float64**

**>>> data.values**

array([0.25, 0.5 , 0.75, 1. , 1.25])

**>>> data.index**

RangeIndex(start=0, stop=5, step=1)

**>>> data[1]**

0.5

**>>> data[2]**

0.75

**>>> data[0:2]**

0 0.25

1 0.50

**dtype: float64**

**>>> data[::-1]**

4 1.25

3 1.00

2 0.75

1 0.50

0 0.25

**dtype: float64**

**>>> data**

0 0.25

**1 0.50**

**2 0.75**

**3 1.00**

**4 1.25**

**dtype: float64**

**>>>**

**Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the NumPy array has an *implicitly defined* integer index used to access the values, the Pandas Series has an *explicitly defined* index associated with the values.**

**This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:**

**>>> data=pd.Series([1,2,3,4,5,6], index=['a','b','c','d','e','f'])**

>>> data

a 1

b 2

c 3

d 4

e 5

f 6

dtype: int64

**We can use non sequential indices as follows**

**>>> data=pd.Series([1,2,3,4,5,6],index=[2,3,4,5,6,7])**

**>>> data**

**2 1**

**3 2**

**4 3**

**5 4**

**6 5**

**7 6**

**dtype: int64**

**>>>**

**For example, data can be a list or NumPy array, in which case index defaults to an**

**integer sequence:**

**In[14]: pd.Series([2, 4, 6])**

**Out[14]: 0 2**

**1 4**

**2 6**

**dtype: int64**

**data can be a scalar, which is repeated to fill the specified index:**

**In[15]: pd.Series(5, index=[100, 200, 300])**

**Out[15]: 100 5**

**200 5**

**300 5**

**dtype: int64**

**data can be a dictionary, in which index defaults to the sorted dictionary keys:**

**In[16]: pd.Series({2:'a', 1:'b', 3:'c'})**

**Out[16]: 1 b**

**2 a**

**3 c**

**dtype: object**

**You can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure that maps typed keys to a set of typed values.**

**This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.**

**We can make the Series-as-dictionary analogy even more clear by constructing a Series object directly from a Python dictionary:**

**pop\_dict={'salem':32143,'coimbatore':321443,'erode':43243,'madurai':43243,'tirunelveli':3424342}**

**>>> data=pd.Series(pop\_dict)**

**>>> data**

**coimbatore 321443**

**erode 43243**

**madurai 43243**

**salem 32143**

**tirunelveli 3424342**

**dtype: int64**

**Unlike a dictionary, though, the Series also supports array-style operations such as slicing:**

**>>> data['coimbatore':'salem']**

**coimbatore 321443**

**erode 43243**

**madurai 43243**

**salem 32143**

**dtype: int64**

Pandas DataFrame Object

**Series is an analog of a one-dimensional array with flexible indices, a DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.**

**>>> population={'Salem':4234,'Erode':32423,'Namakkal':3124,'Karur':32323}**

**>>> area={'Salem':3234,'Erode':5423,'Namakkal':4124,'Karur':62323}**

**>>> data=pd.DataFrame({'Population':population,'Area':area})**

**>>> data**

**Area Population**

**Erode 5423 32423**

**Karur 62323 32323**

**Namakkal 4124 3124**

**Salem 3234 4234**

**Like the Series object, the DataFrame has an index attribute that gives access to the**

**index labels:**

**>>> data.index**

**Index(['Erode', 'Karur', 'Namakkal', 'Salem'], dtype='object')**

**Additionally, the DataFrame has a columns attribute, which is an Index object holding**

**the column labels:**

**>>> data.columns**

**Index(['Area', 'Population'], dtype='object')**

**>>>**

**Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data**

**Example-2**

**>>> sathish={'Tamil':32,'English':32,'Maths':54,'Science':65,'History':43}**

**>>> siva={'Tamil':62,'English':52,'Maths':74,'Science':85,'History':45}**

**>>> santhosh={'Tamil':82,'English':72,'Maths':54,'Science':85,'History':85}**

**>>> data=pd.DataFrame({'Sathish':sathish,'Siva':siva,'Santhosh':santhosh})**

**>>> data**

**Santhosh Sathish Siva**

**English 72 32 52**

**History 85 43 45**

**Maths 54 54 74**

**Science 85 65 85**

**Tamil 82 32 62**

**>>> data.index**

**Index(['English', 'History', 'Maths', 'Science', 'Tamil'], dtype='object')**

**>>> data.columns**

**Index(['Santhosh', 'Sathish', 'Siva'], dtype='object')**

**>>> data['Santhosh']**

**English 72**

**History 85**

**Maths 54**

**Science 85**

**Tamil 82**

**Name: Santhosh, dtype: int64**

**>>> data['Sathish']**

**English 32**

**History 43**

**Maths 54**

**Science 65**

**Tamil 32**

**Name: Sathish, dtype: int64**

**1>>>**

DataFrame Basic Functionality.

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Attribute or Method** | **Description** |
| 1 | T | Transposes rows and columns. |
| 2 | axes | Returns a list with the row axis labels and column axis labels as the only members. |
| 3 | dtypes | Returns the dtypes in this object. |
| 4 | empty | True if NDFrame is entirely empty [no items]; if any of the axes are of length 0. |
| 5 | ndim | Number of axes / array dimensions. |
| 6 | shape | Returns a tuple representing the dimensionality of the DataFrame. |
| 7 | size | Number of elements in the NDFrame. |
| 8 | values | Numpy representation of NDFrame. |
| 9 | head() | Returns the first n rows. |
| 10 | tail() | Returns last n rows. |

**>>> x=pd.DataFrame({"Name":["Sathish","Dr.Balu","Arun","Parthiban"],"Age":[41,34,32,32],"City":["Salem","Salem","Salem","Salem"],"Desig":["Trainer","Engg","Developer","Developer"]},columns=["Name","Age","City","Desig"])**

**>>> x**

**Name Age City Desig**

**0 Sathish 41 Salem Trainer**

**1 Dr.Balu 34 Salem Engg**

**2 Arun 32 Salem Developer**

**3 Parthiban 32 Salem Developer**

**>>> x.values**

**array([['Sathish', 41, 'Salem', 'Trainer'],**

**['Dr.Balu', 34, 'Salem', 'Engg'],**

**['Arun', 32, 'Salem', 'Developer'],**

**['Parthiban', 32, 'Salem', 'Developer']], dtype=object)**

**>>> x.index**

**RangeIndex(start=0, stop=4, step=1)**

**>>> x.T**

**0 1 2 3**

**Name Sathish Dr.Balu Arun Parthiban**

**Age 41 34 32 32**

**City Salem Salem Salem Salem**

**Desig Trainer Engg Developer Developer**

**>>> x.axes**

[RangeIndex(start=0, stop=4, step=1), Index(['Name', 'Age', 'City', 'Desig'], dtype='object')]

>>> x.dtypes

Name object

Age int64

City object

Desig object

dtype: object

**>>> x.empty**

False

**>>> x.ndim**

2

**>>> x.shape**

(4, 4)

**>>> x.values**

**array([['Sathish', 41, 'Salem', 'Trainer'],**

**['Dr.Balu', 34, 'Salem', 'Engg'],**

**['Arun', 32, 'Salem', 'Developer'],**

**['Parthiban', 32, 'Salem', 'Developer']], dtype=object)**

**>>> x.head()**

Name Age City Desig

0 Sathish 41 Salem Trainer

1 Dr.Balu 34 Salem Engg

2 Arun 32 Salem Developer

3 Parthiban 32 Salem Developer

**>>> x.tail()**

Name Age City Desig

0 Sathish 41 Salem Trainer

1 Dr.Balu 34 Salem Engg

2 Arun 32 Salem Developer

3 Parthiban 32 Salem Developer

**>>> x.tail(1)**

Name Age City Desig

3 Parthiban 32 Salem Developer

**>>> x.tail(2)**

Name Age City Desig

2 Arun 32 Salem Developer

3 Parthiban 32 Salem Developer

**>>> x.head(2)**

**Name Age City Desig**

0 Sathish 41 Salem Trainer

1 Dr.Balu 34 Salem Engg

>>>

Let us now understand the functions under Descriptive Statistics in Python Pandas. The following table list down the important functions –

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Function** | **Description** |
| 1 | count() | Number of non-null observations |
| 2 | sum() | Sum of values |
| 3 | mean() | Mean of Values |
| 4 | median() | Median of Values |
| 5 | mode() | Mode of values |
| 6 | std() | Standard Deviation of the Values |
| 7 | min() | Minimum Value |
| 8 | max() | Maximum Value |
| 9 | abs() | Absolute Value |
| 10 | prod() | Product of Values |
| 11 | cumsum() | Cumulative Sum |
| 12 | cumprod() | Cumulative Product |

**>>> x.sum()**

**Name SathishDr.BaluArunParthiban**

**Age 139**

**City SalemSalemSalemSalem**

**Desig TrainerEnggDeveloperDeveloper**

**dtype: object**

**>>> x.sum(1)**

**0 41**

**1 34**

**2 32**

**3 32**

**dtype: int64**

**>>> x.mean()**

**Age 34.75**

**dtype: float64**

**>>> x.sum()**

**Name SathishDr.BaluArunParthiban**

**Age 139**

**City SalemSalemSalemSalem**

**Desig TrainerEnggDeveloperDeveloper**

**dtype: object**

**>>> x.sum()**

**Name SathishDr.BaluArunParthiban**

**Age 139**

**City SalemSalemSalemSalem**

**Desig TrainerEnggDeveloperDeveloper**

**dtype: object**

**>>> x.mean()**

**Age 34.75**

**dtype: float64**

**>>> x.std()**

**Age 4.272002**

**dtype: float64**

**>>> x.count()**

**Name 4**

**Age 4**

**City 4**

**Desig 4**

**dtype: int64**

**>>> x.median()**

Age 33.0

dtype: float64

**>>> x.cumsum()**

Name Age City \

0 Sathish 41 Salem

1 SathishDr.Balu 75 SalemSalem

2 SathishDr.BaluArun 107 SalemSalemSalem

3 SathishDr.BaluArunParthiban 139 SalemSalemSalemSalem

Desig

0 Trainer

1 TrainerEngg

2 TrainerEnggDeveloper

3 TrainerEnggDeveloperDeveloper

Since DataFrame is a Heterogeneous data structure. Generic operations don’t work with all functions.

* Functions like **sum(), cumsum()** work with both numeric and character (or) string data elements without any error. Though **n**practice, character aggregations are never used generally, these functions do not throw any exception.
* Functions like **abs(), cumprod()** throw exception when the DataFrame contains character or string data because such operations cannot be performed.

The **describe()** function computes a summary of statistics pertaining to the DataFrame columns.

>>> **x.describe()**

Age

count 4.000000

mean 34.750000

std 4.272002

min 32.000000

25% 32.000000

50% 33.000000

75% 35.750000

max 41.000000

>>>

**Indexing**

**>>> data=pd.DataFrame({"Area":["Salem","Erode","Coimbatore","Palakad","Trisur"],"Population":[31323,45354,65656,45555,764343]})**

**>>> data**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

4 Trisur 764343

**>>> data.Area**

0 Salem

1 Erode

2 Coimbatore

3 Palakad

4 Trisur

Name: Area, dtype: object

**>>> data.Population**

0 31323

1 45354

2 65656

3 45555

4 764343

Name: Population, dtype: int64

**>>> data['Area']**

0 Salem

1 Erode

2 Coimbatore

3 Palakad

4 Trisur

Name: Area, dtype: object

**>>> data[['Area','Population']]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

4 Trisur 764343

**>>> data.Area is data['Area']**

**True**

**>>> data[0:4]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

Thus for array-style indexing, we need another convention. Here Pandas again uses the loc, iloc, and ix indexers mentioned earlier. Using the iloc indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the DataFrame index and column labels are maintained in the result:

Access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based

**>>> data.loc[0:4]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

4 Trisur 764343

**>>> data.iloc[0:4]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

**>>> data.iloc[:2,:2]**

Area Population

0 Salem 31323

1 Erode 45354

**>>> data.iloc[:3,:2]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

**>>> data.iloc[:3,:1]**

Area

0 Salem

1 Erode

2 Coimbatore

**>>> data.ix[:3,:2]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

**>>> data.ix[:1,:2]**

Area Population

0 Salem 31323

1 Erode 45354

**>>> data.ix[:0,:2]**

Area Population

0 Salem 31323

**>>> data.ix[:3,:2]**

Area Population

0 Salem 31323

1 Erode 45354

2 Coimbatore 65656

3 Palakad 45555

**>>>**

**in the loc indexer we can combine masking and fancy indexing as in the following:**

**>>> data.loc[data.Area>5000,['Population','city']]**

Population city

0 31323 Salem

1 45354 Erode

**Masking**

**>>> data[(data.Area>5000) &(data.Population>3000)]**

**city Population Area density**

**0 Salem 31323 321342 10.258979**

**1 Erode 45354 23413 0.516228**

**Masking and Fancy Indexing**

**>>> data.loc[(data.Area>5000)&(data.Population>3000),['city']]**

city

0 Salem

1 Erode

**>>> data.loc[(data.Area>5000)&(data.Population>3000),['city','Area','Population']]**

city Area Population

0 Salem 321342 31323

1 Erode 23413 45354

**>>> data.loc[(data.Area>5000)&(data.Population>3000)]**

city Population Area density

0 Salem 31323 321342 10.258979

1 Erode 45354 23413 0.516228

**>>> data.loc[(data.Area>5000)&(data.Population>3000),['city','Area','Population']]**

city Area Population

0 Salem 321342 31323

1 Erode 23413 45354

**Pandas - Reindexing**

**data=pd.DataFrame({"http\_Status":[200,342,423,324],"Response\_time":[0.1,.34,.54,.543]},index=["chrome","edge","firefox","safari"])**

**Combaining DataSet**

**Merge**

**One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction.**

**Categories of Joins**

**The pd.merge() function implements a number of types of joins: the one-to-one, many-to-one, and many-to-many joins. All three types of joins are accessed via an identical call to the pd.merge() interface**;

**One-to-one joins**

Perhaps the simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation.

>>> import pandas as pd

>>> import numpy as np

**d1=pd.DataFrame({"Employee":["Sathish","Sivam","Balu","Arun"],"Dept":["cse","eee","cse","mba"]})**

**d2=pd.DataFrame({"Employee":["Sathish","Sivam","Balu","Arun"],"YOJ":['2018','2017','2016','2000']})**

**>>> d1**

**Dept Employee**

0 cse Sathish

1 eee Sivam

2 cse Balu

3 mba Arun

**>>> d2**

Employee YOJ

0 Sathish 2018

1 Sivam 2017

2 Balu 2016

3 Arun 2000

**>>> d3=pd.merge(d1,d2)**

>>> d3

Dept Employee YOJ

0 cse Sathish 2018

1 eee Sivam 2017

2 cse Balu 2016

3 mba Arun 2000

The **pd.merge()** function recognizes that each DataFrame has an employee” column, and automatically joins using this column as a key. The result of the merge is a new DataFrame that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the “employee” column differs between d1 and d2, and the pd.merge() function correctly accounts for this

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

**>>> d4=pd.DataFrame({"Dept":['cse','eee','mba'],"Admin":['PSK','Kar','Vv']})**

**>>> d4**

**Admin Dept**

**0 PSK cse**

**1 Kar eee**

**2 Vv mba**

**>>> pd.merge(d3,d4)**

**Dept Employee YOJ Admin**

**0 cse Sathish 2018 PSK**

**1 cse Balu 2016 PSK**

**2 eee Sivam 2017 Kar**

**3 mba Arun 2000 Vv**

The resulting Data Frame has an additional column with the “Admin” information, where the information is repeated in one or more locations as required by the inputs.

**Many-to-many joins**

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group.

**d5=pd.DataFrame({"Dept":['cse','eee','mba','cse','mba','eee'],"Skill\_Set":['compiler','electricalckt','hr','cloud','finance','medicalinstuments']})**

>>> d5

Dept Skill\_Set

0 cse compiler

1 eee electricalckt

2 mba hr

3 cse cloud

4 mba finance

5 eee medicalinstuments

**>>> pd.merge(d3,d5)**

Dept Employee YOJ Skill\_Set

0 cse Sathish 2018 compiler

1 cse Sathish 2018 cloud

2 cse Balu 2016 compiler

3 cse Balu 2016 cloud

4 eee Sivam 2017 electricalckt

5 eee Sivam 2017 medicalinstuments

6 mba Arun 2000 hr

7 mba Arun 2000 finance

>>>

**Joins**

**>>> state=pd.DataFrame({"state\_name":["Tamilnadu","Andhra","karnataka","kerala","Westbengal"],"State\_id":[1,2,3,4,5]})**

**>>> state**

State\_id state\_name

0 1 Tamilnadu

1 2 Andhra

2 3 karnataka

3 4 kerala

4 5 Westbengal

**place=pd.DataFrame({"Name\_of\_place":["Salem","Coimbatore","Ernakulam","Hyderabad","Mumbai","Mysore"],"State\_id":[1,1,4,2,6,3]})**

>>>

**>>> pd.merge(place,state)**

Name\_of\_place State\_id state\_name

0 Salem 1 Tamilnadu

1 Coimbatore 1 Tamilnadu

2 Ernakulam 4 kerala

3 Hyderabad 2 Andhra

4 Mysore 3 karnataka

**>>> pd.merge(place,state,on='State\_id')**

Name\_of\_place State\_id state\_name

0 Salem 1 Tamilnadu

1 Coimbatore 1 Tamilnadu

2 Ernakulam 4 kerala

3 Hyderabad 2 Andhra

4 Mysore 3 karnataka

**>>> pd.merge(place,state,on='State\_id',how='right')**

Name\_of\_place State\_id state\_name

0 Salem 1 Tamilnadu

1 Coimbatore 1 Tamilnadu

2 Ernakulam 4 kerala

3 Hyderabad 2 Andhra

4 Mysore 3 karnataka

5 NaN 5 Westbengal

**>>> pd.merge(place,state,on='State\_id',how='left')**

Name\_of\_place State\_id state\_name

0 Salem 1 Tamilnadu

1 Coimbatore 1 Tamilnadu

2 Ernakulam 4 kerala

3 Hyderabad 2 Andhra

4 Mumbai 6 NaN

5 Mysore 3 karnataka

**>>> pd.merge(place,state,on='State\_id',how='outer')**

Name\_of\_place State\_id state\_name

0 Salem 1 Tamilnadu

1 Coimbatore 1 Tamilnadu

2 Ernakulam 4 kerala

3 Hyderabad 2 Andhra

4 Mumbai 6 NaN

5 Mysore 3 karnataka

6 NaN 5 Westbengal

**>>> pd.merge(place,state,on='State\_id',how='inner')**

Name\_of\_place State\_id state\_name

0 Salem 1 Tamilnadu

1 Coimbatore 1 Tamilnadu

2 Ernakulam 4 kerala

3 Hyderabad 2 Andhra

4 Mysore 3 karnataka

>>>

**Pandas was developed in the context of financial modelling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and timeindexed data. Date and time data comes in a few flavors, which we will discuss here**

***Time stamps*** reference particular moments in time (e.g., July 4th, 2015, at 7:00a.m.).

***Time intervals* and *periods***reference a length of time between a particular beginning and end point—for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods constituting days).

***Time deltas* or *durations*** reference an exact length of time (e.g., a duration of 22.56 seconds).

**Dates and times in Pandas: Best of both worlds Pandas builds upon all the tools just discussed to provide a Timestamp object, which combines the ease of use of datetime and dateutil with the efficient storage andvectorized interface of numpy.datetime64. From a group of these Timestamp objects, Pandas can construct a DatetimeIndex that can be used to index data in a Series or DataFrame;**

>>> import pandas as pd

>>> import numpy as np

>>> date=**pd.to\_datetime("jul 6th,2018")**

>>> date

Timestamp('2018-07-06 00:00:00')

**>>> date.strftime("%A")**

'Friday'

**>>> date+pd.to\_timedelta(np.arange(12))**

DatetimeIndex([ '2018-07-06', '2018-07-06', '2018-07-06', '2018-07-06',

'2018-07-06', '2018-07-06', '2018-07-06', '2018-07-06',

'2018-07-06', '2018-07-06', '2018-07-06', '2018-07-06'],

dtype='datetime64[ns]', freq=None)

>>>

>>> import pandas as pd

>>> dates=pd.date\_range('1/1/2018',periods=10)

>>> dates

DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',

'2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',

'2018-01-09', '2018-01-10'],

dtype='datetime64[ns]', freq='D')

>>> data=pd.DataFrame(np.random.randn(10,4))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'np' is not defined

>>> import numpy as np

>>> data=pd.DataFrame(np.random.randn(10,4))

>>> data

0 1 2 3

0 0.638571 0.478564 -2.521931 -0.318697

1 -0.864927 -0.573170 0.101631 1.761234

2 2.255759 0.523141 0.200565 1.145966

3 -0.617701 -0.360860 0.474360 0.687512

4 -0.647126 1.394531 -2.289070 0.548037

5 -0.788754 0.757415 0.489004 0.993815

6 -1.041785 -0.997872 0.602224 0.577130

7 -1.057408 -0.553987 -0.716445 0.114301

8 1.422278 0.301021 1.405026 0.080285

9 2.166691 0.835413 0.172597 -0.982091

>>> data=pd.DataFrame(np.random.randn(10,4),index=dates)

>>> data

0 1 2 3

2018-01-01 -1.382382 -1.312808 0.580930 -0.741043

2018-01-02 -0.534516 0.432655 -1.353367 1.511684

2018-01-03 -1.533125 -0.140758 -0.704485 -0.380711

2018-01-04 0.297457 -0.665140 0.057138 -0.171316

2018-01-05 1.028022 0.396086 0.323962 -0.116355

2018-01-06 -1.298905 0.444092 0.095835 0.269598

2018-01-07 -0.068647 0.771054 -0.206817 1.268032

2018-01-08 -0.438859 0.926416 0.921184 -1.792419

2018-01-09 -0.079481 -0.089264 0.072859 -1.187843

2018-01-10 1.829377 0.591865 0.346438 0.630595

>>> data=pd.DataFrame(np.random.randn(10,4),index=dates,columns=['A','B','C','D'])

>>> data

A B C D

2018-01-01 0.986593 -1.431859 -1.248647 -0.470785

2018-01-02 -0.278658 -0.571776 0.408524 -0.566320

2018-01-03 0.034232 1.211142 0.560120 0.435981

2018-01-04 -1.498934 0.217521 -0.550173 -1.752011

2018-01-05 -0.476367 -1.640955 -1.224929 1.348408

2018-01-06 -1.558814 -0.816513 0.372841 0.774528

2018-01-07 -0.876119 -1.344271 -0.458426 0.691965

2018-01-08 1.827171 -0.334706 1.525819 -0.329380

2018-01-09 -0.597232 -0.181605 -0.375745 0.838083

2018-01-10 -0.691415 -1.423166 1.300285 1.699993

>>> data.isnull()

A B C D

2018-01-01 False False False False

2018-01-02 False False False False

2018-01-03 False False False False

2018-01-04 False False False False

2018-01-05 False False False False

2018-01-06 False False False False

2018-01-07 False False False False

2018-01-08 False False False False

2018-01-09 False False False False

2018-01-10 False False False False

**# second represenation of date in pandas**

>>> index=pd.DatetimeIndex(['2018-09-01','2018-09-02','2018-09-03','2018-09-04','2018-09-05','2018-09-06','2018-09-07','2018-09-08','2018-09-09','2018-09-10',])

>>> index

DatetimeIndex(['2018-09-01', '2018-09-02', '2018-09-03', '2018-09-04',

'2018-09-05', '2018-09-06', '2018-09-07', '2018-09-08',

'2018-09-09', '2018-09-10'],

dtype='datetime64[ns]', freq=None)

>>> data=pd.DataFrame(np.random.randn(10,5),index=index)

>>> data

0 1 2 3 4

2018-09-01 0.993403 -0.578916 -2.664258 -0.992262 0.276268

2018-09-02 0.404130 0.003154 -1.410321 -0.140580 -0.786281

2018-09-03 -0.503353 0.030555 1.403806 1.117911 -1.413224

2018-09-04 0.375304 0.233901 -0.232896 1.104111 -0.955210

2018-09-05 -1.696118 -0.648601 1.391220 0.201426 0.820210

2018-09-06 0.383951 0.794670 -1.026958 -1.332138 -0.806718

2018-09-07 -0.333007 -0.244515 -0.301350 1.221619 -0.665142

2018-09-08 -0.419681 0.531021 0.750632 0.073427 -1.709533

2018-09-09 0.992748 2.061857 -0.773238 -0.769559 -0.369897

2018-09-10 1.723965 0.065919 -1.330857 -0.533415 -0.473009

>>>

**>>> data['2018-09-01':'2018-09-04']**

0 1 2 3 4

2018-09-01 0.993403 -0.578916 -2.664258 -0.992262 0.276268

2018-09-02 0.404130 0.003154 -1.410321 -0.140580 -0.786281

2018-09-03 -0.503353 0.030555 1.403806 1.117911 -1.413224

2018-09-04 0.375304 0.233901 -0.232896 1.104111 -0.955210

>>>

**>>> data['2018']**

0 1 2 3 4

2018-09-01 0.993403 -0.578916 -2.664258 -0.992262 0.276268

2018-09-02 0.404130 0.003154 -1.410321 -0.140580 -0.786281

2018-09-03 -0.503353 0.030555 1.403806 1.117911 -1.413224

2018-09-04 0.375304 0.233901 -0.232896 1.104111 -0.955210

2018-09-05 -1.696118 -0.648601 1.391220 0.201426 0.820210

2018-09-06 0.383951 0.794670 -1.026958 -1.332138 -0.806718

2018-09-07 -0.333007 -0.244515 -0.301350 1.221619 -0.665142

2018-09-08 -0.419681 0.531021 0.750632 0.073427 -1.709533

2018-09-09 0.992748 2.061857 -0.773238 -0.769559 -0.369897

2018-09-10 1.723965 0.065919 -1.330857 -0.533415 -0.473009

**>>> data['2018-09-01':'2018-09-04']**

0 1 2 3 4

2018-09-01 0.993403 -0.578916 -2.664258 -0.992262 0.276268

2018-09-02 0.404130 0.003154 -1.410321 -0.140580 -0.786281

2018-09-03 -0.503353 0.030555 1.403806 1.117911 -1.413224

2018-09-04 0.375304 0.233901 -0.232896 1.104111 -0.955210

**>>> dates=pd.date\_range('1/1/2018','20/1/2018')**

>>> dates

DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',

'2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',

'2018-01-09', '2018-01-10', '2018-01-11', '2018-01-12',

'2018-01-13', '2018-01-14', '2018-01-15', '2018-01-16',

'2018-01-17', '2018-01-18', '2018-01-19', '2018-01-20'],

dtype='datetime64[ns]', freq='D')

**>>> dates=pd.date\_range('1-1-2018','20-1-2018')**

>>> dates

DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',

'2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',

'2018-01-09', '2018-01-10', '2018-01-11', '2018-01-12',

'2018-01-13', '2018-01-14', '2018-01-15', '2018-01-16',

'2018-01-17', '2018-01-18', '2018-01-19', '2018-01-20'],

dtype='datetime64[ns]', freq='D')

**>>> dates=pd.date\_range('1-1-2018',periods=20)**

>>> dates

DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',

'2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',

'2018-01-09', '2018-01-10', '2018-01-11', '2018-01-12',

'2018-01-13', '2018-01-14', '2018-01-15', '2018-01-16',

'2018-01-17', '2018-01-18', '2018-01-19', '2018-01-20'],

dtype='datetime64[ns]', freq='D')

**>>> dates=pd.date\_range('1-1-2018',periods=20,freq=H)**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'H' is not defined

**>>> dates=pd.date\_range('1-1-2018',periods=20,freq='H')**

>>> dates

DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 01:00:00',

'2018-01-01 02:00:00', '2018-01-01 03:00:00',

'2018-01-01 04:00:00', '2018-01-01 05:00:00',

'2018-01-01 06:00:00', '2018-01-01 07:00:00',

'2018-01-01 08:00:00', '2018-01-01 09:00:00',

'2018-01-01 10:00:00', '2018-01-01 11:00:00',

'2018-01-01 12:00:00', '2018-01-01 13:00:00',

'2018-01-01 14:00:00', '2018-01-01 15:00:00',

'2018-01-01 16:00:00', '2018-01-01 17:00:00',

'2018-01-01 18:00:00', '2018-01-01 19:00:00'],

dtype='datetime64[ns]', freq='H')

**>>> time=pd.timedelta\_range(0,periods=10,freq='H')**

>>> time

TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',

'05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],

dtype='timedelta64[ns]', freq='H')

**>>> dates=pd.date\_range('1-1-2018',periods=20,freq='D')**

>>> dates

DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',

'2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',

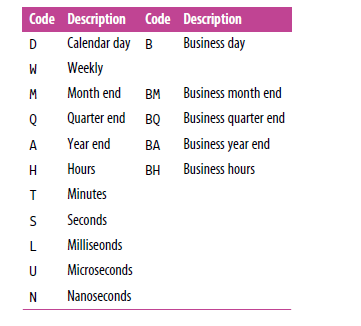
'2018-01-09', '2018-01-10', '2018-01-11', '2018-01-12',

'2018-01-13', '2018-01-14', '2018-01-15', '2018-01-16',

'2018-01-17', '2018-01-18', '2018-01-19', '2018-01-20'],

dtype='datetime64[ns]', freq='D')

>>>

****

**>>> time=pd.timedelta\_range(0,periods=10,freq='2H30T')**

>>> time

TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',

'12:30:00', '15:00:00', '17:30:00', '20:00:00', '22:30:00'],

dtype='timedelta64[ns]', freq='150T')

>>>

**Renaming the Columns of Pandas Data Frame**

Microsoft Windows [Version 10.0.17134.112]

(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\I18N>python

Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> import pandas as pd

>>> data=pd.read\_html("https://en.wikipedia.org/wiki/National\_Institutes\_of\_Technology",header=0)[1]

>>> data

**Serial No Name Short Name ... Established City/Town State/UT**

0 1 NIT Allahabad MNNIT ... 2001 Allahabad Uttar Pradesh

1 2 NIT Bhopal MANIT ... 2002 Bhopal Madhya Pradesh

2 3 NIT Calicut NITC ... 2002 Calicut Kerala

3 4 NIT Hamirpur NITH ... 2002 Hamirpur Himachal Pradesh

4 5 NIT Jaipur MNIT ... 2002 Jaipur Rajasthan

...

30 31 NIT Andhra Pradesh NITANP ... 2015 Tadepalligudem Andhra Pradesh

[31 rows x 7 columns]

**>>> data.columns**

Index(['Serial No', 'Name', 'Short Name', 'Founded', 'Established', 'City/Town', 'State/UT'], dtype='object')

>>> data.rename(columns={"Serial No":"S.No","Name":"Name\_Institution"},inplace=True)

>>> data

S.No Name\_Institution Short Name Founded Established City/Town State/UT

0 1 NIT Allahabad MNNIT 1961 2001 Allahabad Uttar Pradesh

1 2 NIT Bhopal MANIT 1960 2002 Bhopal Madhya Pradesh

2 3 NIT Calicut NITC 1961 2002 Calicut Kerala

3 4 NIT Hamirpur NITH 1986 2002 Hamirpur Himachal Pradesh

**>>> data.columns**

Index(['S.No', 'Name\_Institution', 'Short Name', 'Founded', 'Established',

'City/Town', 'State/UT'],

dtype='object')

**Alternative Approach**

>>> data\_col=["Serial","Institutions Name","Abrevation","Founded\_year","Established\_year","City","State"]

>>> data.columns=data\_col

>>> data.head()

Serial Institutions Name Abrevation Founded\_year Established\_year City State

0 1 NIT Allahabad MNNIT 1961 2001 Allahabad Uttar Pradesh

1 2 NIT Bhopal MANIT 1960 2002 Bhopal Madhya Pradesh

2 3 NIT Calicut NITC 1961 2002 Calicut Kerala

3 4 NIT Hamirpur NITH 1986 2002 Hamirpur Himachal Pradesh

4 5 NIT Jaipur MNIT 1963 2002 Jaipur Rajasthan

>>>

**Dropping the Column in Pandas**

**>>> import pandas as pd**

**>>> data=pd.read\_html("https://en.wikipedia.org/wiki/National\_Institutes\_of\_Technology",header=0)[1]**

**>>> data.info()**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 31 entries, 0 to 30

Data columns (total 7 columns):

Serial No 31 non-null int64

Name 31 non-null object

Short Name 31 non-null object

Founded 31 non-null int64

Established 31 non-null int64

City/Town 31 non-null object

State/UT 31 non-null object

dtypes: int64(3), object(4)

memory usage: 1.8+ KB

**>>> data.drop('Founded',axis=1)**

**Serial No Name Short Name Established City/Town State/UT**

0 1 NIT Allahabad MNNIT 2001 Allahabad Uttar Pradesh

1 2 NIT Bhopal MANIT 2002 Bhopal Madhya Pradesh

2 3 NIT Calicut NITC 2002 Calicut Kerala

3 4 NIT Hamirpur NITH 2002 Hamirpur Himachal Pradesh

4 5 NIT Jaipur MNIT 2002 Jaipur Rajasthan

5 6 NIT Jalandhar NITJ 2002 Jalandhar Punjab

**Note that in the above dataframe we don’t have Founded Columns**

**>>> data.info()**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 31 entries, 0 to 30

Data columns (total 7 columns):

Serial No 31 non-null int64

Name 31 non-null object

Short Name 31 non-null object

**Founded 31 non-null int64**

Established 31 non-null int64

City/Town 31 non-null object

State/UT 31 non-null object

dtypes: int64(3), object(4)

memory usage: 1.8+ KB

>>> data.drop('Founded',axis=1,**inplace=True**)

>>> data.head()

Serial No Name Short Name Established City/Town State/UT

0 1 NIT Allahabad MNNIT 2001 Allahabad Uttar Pradesh

1 2 NIT Bhopal MANIT 2002 Bhopal Madhya Pradesh

2 3 NIT Calicut NITC 2002 Calicut Kerala

3 4 NIT Hamirpur NITH 2002 Hamirpur Himachal Pradesh

4 5 NIT Jaipur MNIT 2002 Jaipur Rajasthan

>>> data.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 31 entries, 0 to 30

Data columns (total 6 columns):

Serial No 31 non-null int64

Name 31 non-null object

Short Name 31 non-null object

Established 31 non-null int64

City/Town 31 non-null object

State/UT 31 non-null object

dtypes: int64(2), object(4)

memory usage: 1.5+ KB

>>>

*# dropna with how='any' would drop any row with 'NaN'*

**>>> data=pd.read\_csv("http://bit.ly/uforeports")**

**>>> data.info()**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 18241 entries, 0 to 18240

Data columns (total 5 columns):

City 18216 non-null object

Colors Reported 2882 non-null object

Shape Reported 15597 non-null object

State 18241 non-null object

Time 18241 non-null object

dtypes: object(5)

memory usage: 712.6+ KB

**>>> data.head()**

City Colors Reported Shape Reported State Time

0 Ithaca NaN TRIANGLE NY 6/1/1930 22:00

1 Willingboro NaN OTHER NJ 6/30/1930 20:00

2 Holyoke NaN OVAL CO 2/15/1931 14:00

3 Abilene NaN DISK KS 6/1/1931 13:00

4 New York Worlds Fair NaN LIGHT NY 4/18/1933 19:00

**>>> data.dropna(how="any").head()**

City Colors Reported Shape Reported State Time

12 Belton RED SPHERE SC 6/30/1939 20:00

19 Bering Sea RED OTHER AK 4/30/1943 23:00

36 Portsmouth RED FORMATION VA 7/10/1945 1:30

44 Blairsden GREEN SPHERE CA 6/30/1946 19:00

82 San Jose BLUE CHEVRON CA 7/15/1947 21:00

>>>

**Filling with NaN using ffill and bfill methods**

**data**.**fillna(method**=**'bfill')**

**>>> data.drop(2,axis=0).head()**

City Colors Reported Shape Reported State Time

0 Ithaca NaN TRIANGLE NY 6/1/1930 22:00

1 Willingboro NaN OTHER NJ 6/30/1930 20:00

3 Abilene NaN DISK KS 6/1/1931 13:00

4 New York Worlds Fair NaN LIGHT NY 4/18/1933 19:00

5 Valley City NaN DISK ND 9/15/1934 15:30

**>>> data.drop([1,2,3],axis=0).head()**

City Colors Reported Shape Reported State Time

0 Ithaca NaN TRIANGLE NY 6/1/1930 22:00

4 New York Worlds Fair NaN LIGHT NY 4/18/1933 19:00

5 Valley City NaN DISK ND 9/15/1934 15:30

6 Crater Lake NaN CIRCLE CA 6/15/1935 0:00

7 Alma NaN DISK MI 7/15/1936 0:00

**>>>**

**Matplotlib**

Matplotlib is a python library used to create 2D graphs and plots by using python scripts. It has a module named pyplot which makes things easy for plotting by providing feature to control line styles, font properties, formatting axes etc. It supports a very wide variety of graphs and plots namely - histogram, bar charts, power spectra, error charts etc.

It is used along with NumPy to provide an environment that is an effective open source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython.

Conventionally, the package is imported into the Python script by adding the following statement –

from matplotlib import pyplot as plt

If you are using Matplotlib from within a script, the function plt.show() is your friend. plt.show() starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> **x=np.linspace(0,100,10)**

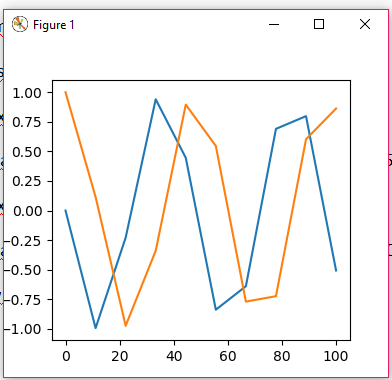
>>> plt.plot(x,np.sin(x))

[<matplotlib.lines.Line2D object at 0x00000266AC908588>]

>>> plt.plot(x,np.cos(x))

[<matplotlib.lines.Line2D object at 0x00000266A977D048>]

>>> plt.show()



The **plt.show()** command does a lot under the hood, as it must interact with your system’s interactive graphical backend. The details of this operation can vary greatly from system to system and even installation to installation, but Matplotlib does its best to hide all these details from you.

One thing to be aware of: the plt.show() command should be used only once per Python session, and is most often seen at the very end of the script. Multiple show() commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

Instead of the linear graph, the values can be displayed discretely by adding a format string to the plot() function. Following formatting characters can be used.

|  |  |
| --- | --- |
| **Sr.No.** | **Character & Description** |
| 1 | **'-'** Solid line style |
| 2 | **'--'** Dashed line style |
| 3 | **'-.'** Dash-dot line style |
| 4 | **':'** Dotted line style |
| 5 | **'.'** Point marker |
| 6 | **','** Pixel marker |
| 7 | **'o'** Circle marker |
| 8 | **'v'** Triangle\_down marker |
| 9 | **'^'** Triangle\_up marker |
| 10 | **'<'** Triangle\_left marker |
| 11 | **'>'** Triangle\_right marker |
| 12 | **'1'** Tri\_down marker |
| 13 | **'2'** Tri\_up marker |
| 14 | **'3'** Tri\_left marker |
| 15 | **'4'** Tri\_right marker |
| 16 | **'s'** Square marker |
| 17 | **'p'** Pentagon marker |
| 18 | **'\*'** Star marker |
| 19 | **'h'** Hexagon1 marker |
| 20 | **'H'** Hexagon2 marker |
| 21 | **'+'** Plus marker |
| 22 | **'x'** X marker |
| 23 | **'D'** Diamond marker |
| 24 | **'d'** Thin\_diamond marker |
| 25 | **'|'** Vline marker |
| 26 | **'\_'** Hline marker |

The following color abbreviations are also defined.

|  |  |
| --- | --- |
| **Character** | **Color** |
| 'b' | Blue |
| 'g' | Green |
| 'r' | Red |
| 'c' | Cyan |
| 'm' | Magenta |
| 'y' | Yellow |
| 'k' | Black |
| 'w' | White |

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface. For example, the following code will probably look quite familiar to MATLAB users

>>> import numpy as np

>>> import matpotlib.pyplot as plt

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ModuleNotFoundError: No module named 'matpotlib'

>>> import matplotlib.pyplot as plt

>>> x=np.linspace(0,100,10)

**>>> plt.subplot(2,1,1) # rows ,columns, panel**

<matplotlib.axes.\_subplots.AxesSubplot object at 0x000001D41194C470>

>>> plt.plot(x,np.sin(x))

[<matplotlib.lines.Line2D object at 0x000001D4095150F0>]

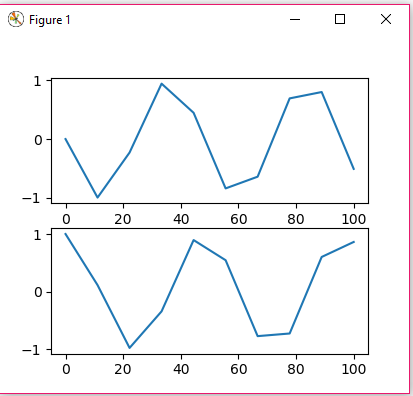
**>>> plt.subplot(2,1,2)**

<matplotlib.axes.\_subplots.AxesSubplot object at 0x000001D414102908>

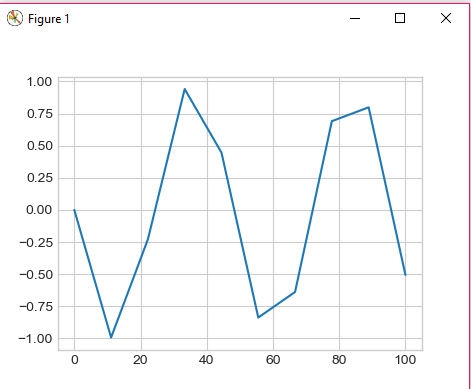
>>> plt.plot(x,np.cos(x) )

[<matplotlib.lines.Line2D object at 0x000001D414102E80>]

>>> plt.show()



>>> plt.style.use('seaborn-whitegrid')



You can adjust the position, size, and style of these labels using optional arguments to the function.When multiple lines are being shown within single axes, it can be useful to create a plot **legend** that labels each line type.

Again, Matplotlib has a built-in way of quicklycreating such a legend. It is done via the (you guessed it) plt.legend() method. Though there are several valid ways of using this, I find it easiest to specify the label of each line using the label keyword of the plot function.

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> x=np.linspace(0,10,1000)

>>> plt.plot(x,np.sin(x),'-g',label='sin(x)')

[<matplotlib.lines.Line2D object at 0x000001A586C6C080>]

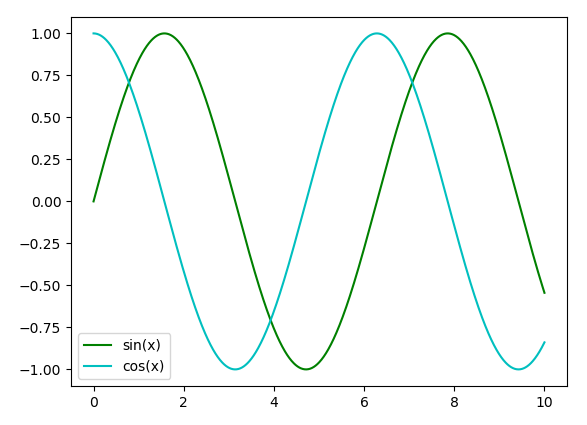
>>> plt.plot(x,np.cos(x),'-c',label='cos(x)')

[<matplotlib.lines.Line2D object at 0x000001A586C6C1D0>]

**>>> plt.legend()**

<matplotlib.legend.Legend object at 0x000001A586C5BEF0>

>>> plt.show()



>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> x=np.linspace(0,10,100)

>>> plt.plot(x,np.sin(x),'-g',label='sin(x)')

[<matplotlib.lines.Line2D object at 0x000001EB56C12860>]

>>> plt.plot(x,np.cos(x),'-c',label='cos(x)')

[<matplotlib.lines.Line2D object at 0x000001EB56C12A90>]

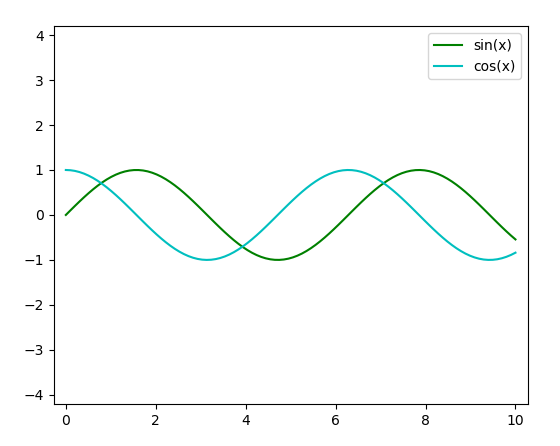
**>>> plt.axis('equal')**

(-0.5, 10.5, -1.0999445244849302, 1.0999973583088063)

**>>> plt.legend()**

<matplotlib.legend.Legend object at 0x000001EB56C12F60>

>>> plt.show()



**Adjusting the plot axis limit**

Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it’s nice to have finer control. The most basic way to adjust axis limits is to use the **plt.xlim() and plt.ylim()** methods

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> x=np.linspace(0,100,1000)

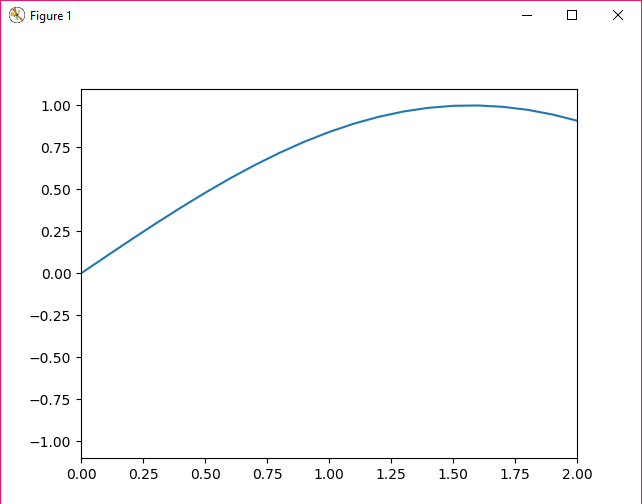
>>> plt.plot(x,np.sin(x))

[<matplotlib.lines.Line2D object at 0x000002B911631860>]

**>>> plt.xlim(0,2)**

(0, 2)

>>> plt.show()



Similarly we have plt.ylim() to limit the y axis.

The plt.axis() method allows you to set the x and y limits with a single call, by passing a list that specifies [xmin, xmax, ymin,ymax]

**Example**

>>> plt.plot(x,np.sin(x))

[<matplotlib.lines.Line2D object at 0x000002B9132A9940>]

>>> plt.axis([0,2,0,3]) # with out using plt.xlim() and plt.ylim()

[0, 2, 0, 3]

>>> plt.show()

It allows even higher-level specifications, such as ensuring an equal aspect ratio so that on your screen, one unit in x is equal to one unit in y.

>>> plt.plot(x,np.sin(x))

[<matplotlib.lines.Line2D object at 0x000002B913160CC0>]

>>> plt.axis('equal')

(-5.0, 105.0, -1.0999999779246914, 1.099999996514425)

>>> plt.show()

In the object-oriented interface to plotting, rather than calling these functions individually,it is often more convenient to use the ax.set() method to set all these properties at once

plt.xlabel() → ax.set\_xlabel()

plt.ylabel() → ax.set\_ylabel()

plt.xlim() → ax.set\_xlim()

plt.ylim() → ax.set\_ylim()

plt.title() → ax.set\_title()

In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the ax.set() method to set all these properties.

at once

In[16]: ax = plt.axes()

ax.plot(x, np.sin(x))

**ax.set(xlim=(0, 10), ylim=(-2, 2),xlabel='x', ylabel='sin(x)',title='A Simple Plot');**

Scatter plot

plt.plot(x, y, '-p', color='gray',

markersize=15, linewidth=4,

markerfacecolor='white',

markeredgecolor='gray',

markeredgewidth=2)

plt.ylim(-1.2, 1.2);

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> x=np.linspace(0,10,30)

>>> y=np.sin(x)

>>> plt.plot(x,y,'ok')

[<matplotlib.lines.Line2D object at 0x0000022AACF218D0>]

>>> plt.show()

>>> plt.scatter(x,y,marker='o',color='k')

<matplotlib.collections.PathCollection object at 0x0000022AAE9D74E0>

>>> plt.show()

>>> **plt.plot(x,y,marker='o',markersize=15,markerfacecolor='white',markeredgecolor='gray',markeredgewidth=2,color='gray')**

[<matplotlib.lines.Line2D object at 0x0000022AAEB7B9B0>]

>>> plt.show()

>>>

The primary difference of **plt.scatter** from **plt.plot** is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

**Random scatter graph**

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> **rang=np.random.RandomState(0)**

>>> x=rang.randn(100)

>>> y=rang.randn(100)

>>> color=rang.rand(100)

>>> size=1000\*rang.rand(100)

>>> plt.scatter(x,y,color,size,alpha=0.3,cmap='viridis')

<matplotlib.collections.PathCollection object at 0x000002631A9B2A58>

>>> plt.colorbar()

<matplotlib.colorbar.Colorbar object at 0x000002631A9F85C0>

>>> plt.show()

>>>

Visualizing a Three-Dimensional Function

We’ll start by demonstrating a contour plot using a function z = f x, y , using the following particular choice for f

A contour plot can be created with the **plt.contour** function. It takes three arguments: a grid of x values, a grid of y values, and a grid of z values. The x and y values represent positions on the plot, and the z values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the **np.meshgrid** function, which builds two-dimensional grids from one-dimensional arrays:

>>> import numpy as np

>>> import matplotlib.pyplot as plt

>>> x=np.linspace(0,5,50)

>>> y=np.linspace(0,4,50)

**>>> def f(x,y):**

**... return np.sin(x)\*\*10 +np.cos(y)**

**...**

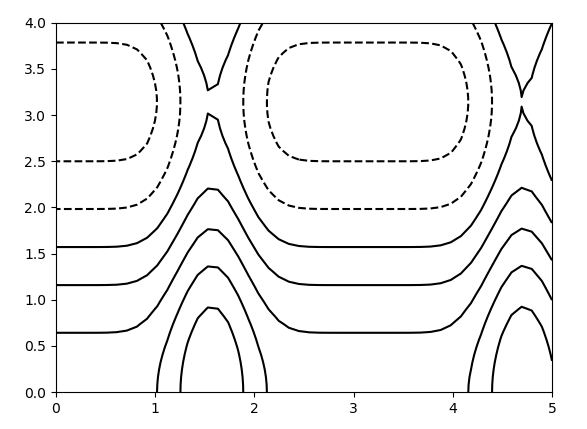
>>> **x,y=np.meshgrid(x,y)**

>>> z=f(x,y)

>>> plt.contour(x,y,z,colors='black')

<matplotlib.contour.QuadContourSet object at 0x0000016C87BA38D0>

>>> plt.show()



Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines.

Alternatively, you can color-code the lines by specifying a colormap with the cmap argument. Here, we’ll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range

>>> plt.contour(x,y,z,20,cmap='RdGy')

<matplotlib.contour.QuadContourSet object at 0x0000016C886B36A0>

>>> plt.show()

Here we chose the RdGy (short for *Red-Gray*) colormap, which is a good choice for centered data.

>>> **plt.contourf** (x,y,z,20,cmap='RdGy')

<matplotlib.contour.QuadContourSet object at 0x0000016C8984CA90>

>>> plt.show()

Histogram

x1 = np.random.normal(0, 0.8, 1000)

x2 = np.random.normal(-2, 1, 1000)

x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True, bins=40)

plt.hist(x1, \*\*kwargs)

plt.hist(x2, \*\*kwargs)

plt.hist(x3, \*\*kwargs);

>>> x=np.arange(0.,5.,.2)

>>> plt.plot(x,x,'r--',x,x\*\*2,'bs',x,x\*\*3,'g^')

[<matplotlib.lines.Line2D object at 0x00000231E77CEE48>, <matplotlib.lines.Line2D object at 0x00000231E77D7048>, <matplotlib.lines.Line2D object at 0x00000231E77D7860>]

>>> plt.show()

Annotation

he [annotate()](https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.annotate) method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument xy and the location of the text xytext. Both of these arguments are (x,y) tuples.

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5), arrowprops=dict(facecolor='black', shrink=0.05))

Working with text

np.random.seed(19680801)

mu, sigma = 100, 15

x = mu + sigma \* np.random.randn(10000)

*# the histogram of the data*

n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')

plt.ylabel('Probability')

plt.title('Histogram of IQ')

plt.text(60, .025, r'$\mu=100,\ \sigma=15$')

plt.axis([40, 160, 0, 0.03])

plt.grid(True)

plt.show()

**BAR Graph**

First, let us understand why do we need a bar graph. A bar graph uses bars to compare data among different categories. It is well suited when you want to measure the changes over a period of time. It can be represented horizontally or vertically. Also, the important thing to keep in mind is that longer the bar, greater is the value. Now, let us practically implement it using python matplotlib.

**Matplotlib Basemap**

The map is created using the Basemap class, which has many options. Without passing any option, the map has the [Plate Carrée projection](http://en.wikipedia.org/wiki/Equirectangular_projection) centered at longitude and latitude = 0

After setting the map, we can draw what we want. In this case, the coast lines layer, which comes already with the library, using the method *drawcoastlines()*

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.basemap import Basemap

m=Basemap()

m.drawcoastlines()

m.fillcontinents(color=’coral’, lake\_color=’aqua’)

Changing the projection is easy, just add the *projection* argument and *lat\_0* and *lon\_0* to the *Basemap* contructor.