7.Python Programs on NumPy Arrays, Linear algebra with NumPy

NumPy arrays:

manual construction of an array using the `np.array` function. The standard convention to import NumPy is as follows:

>>> import numpy as np

>>> p = np.array([48.858598, 2.294495])

>>> p

array([48.858598, 2.294495])

There are two requirements of a NumPy array: a fixed size at creation and a uniform, fixed data type, with a fixed size in memory. The following functions help you to get information on the `p` matrix:

>>> p.ndim   # getting dimension of array p

1

>>> p.shape   # getting size of each array dimension

(2,)

>>> len(p)   # getting dimension length of array p

2

>>> p.dtype   # getting data type of array p

dtype('float64')

Data types:

We can easily convert or cast an array from one `dtype` to another using the `astype` method:

>>> a = np.array([1, 2, 3, 4])

>>> a.dtype

dtype('int64')

>>> float_b = a.astype(np.float64)

>>> float_b.dtype

dtype('float64')

Array creation:

There are various functions provided to create an array object. They are very useful for us to create and store data in a multidimensional array in different situations.

Function	Description	Example

empty, empty_like	Create a new array of the given shape and type, without initializing elements	>>> np.empty([3,2], dtype=np.float64)
		array([[0., 0.], [0., 0.], [0., 0.]])
		>>> a = np.array([[1, 2], [4, 3]])
		>>> np.empty_like(a)
		array([[0, 0], [0, 0]])

eye, identity	Create a NxN identity matrix with ones on the diagonal and zero elsewhere	>>> np.eye(2, dtype=np.int) array([[1, 0], [0, 1]])

ones, ones_like	Create a new array with the given shape and type, filled with 1s for all elements	>>> np.ones(5)
		array([1., 1., 1., 1., 1.])
		>>> np.ones(4, dtype=np.int)
		array([1, 1, 1, 1])
		>>> x = np.array([[0,1,2], [3,4,5]])
		>>> np.ones_like(x)
		array([[1, 1, 1],[1, 1, 1]])

zeros, zeros_like	This is similar to ones, ones_like, but initializing elements with 0s instead	>>> np.zeros(5)
		array([0., 0., 0., 0-, 0.])
		>>> np.zeros(4, dtype=np.int)
		array([0, 0, 0, 0])
		>>> x = np.array([[0, 1, 2], [3, 4, 5]])
		>>> np.zeros_like(x)
		array([[0, 0, 0],[0, 0, 0]])

arange	Create an array with even spaced values in a given interval	>>> np.arange(2, 5) array([2, 3, 4])

		>>> np.arange(4, 12, 5) array([4, 9])

full, full_like	Create a new array with the given shape and type, filled with a selected value	>>> np.full((2,2), 3, dtype=np.int)
		array([[3, 3], [3, 3]])
		>>> x = np.ones(3)
		>>> np.full_like(x, 2)
		array([2., 2., 2.])

array	Create an array from the existing data	>>> np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]]) array([1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])

asarray	Convert the input to an array	>>> a = [3.14, 2.46]
		>>> np.asarray(a)
		array([3.14, 2.46])

copy	Return an array copy of the given object	>>> a = np.array([[1, 2], [3, 4]])
		>>> np.copy(a)
		array([[1, 2], [3, 4]])

fromstring	Create 1-D array from a string or text	>>> np.fromstring('3.14 2.17', dtype=np.float, sep=' ') array([3.14, 2.17])

Indexing and slicing:

As with other Python sequence types, such as lists, it is very easy to access and assign a value of each array's element:

```
>>> a = np.arange(7)
>>> a
array([0, 1, 2, 3, 4, 5, 6])
>>> a[1], a [4], a[-1]
(1, 4, 6)
```

Note

In Python, array indices start at 0. This is in contrast to Fortran or Matlab, where indices begin at 1.

As another example, if our array is multidimensional, we need tuples of integers to index an item:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a[0, 2]     # first row, third column
3
>>> a[0, 2] = 10
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 9]])
>>> b = a[2]
>>> b
array([7, 8, 9])
>>> c = a[:2]
>>> c
array([[1, 2, 10], [4, 5, 6]])
```

We call b and c as array slices, which are views on the original one. It means that the data is not copied to b or c, and whenever we modify their values, it will be reflected in the array a as well:

```
>>> b[-1] = 11
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 11]])
```

Note

We use a colon (:) character to take the entire axis when we omit the index number.

Fancy indexing

Besides indexing with slices, NumPy also supports indexing with Boolean or integer arrays (masks). This method is called **fancy indexing**. It creates copies, not views.

First, we take a look at an example of indexing with a Boolean mask array:

```
>>> a = np.array([3, 5, 1, 10])
>>> b = (a % 5 == 0)
>>> b
array([False, True, False, True], dtype=bool)
>>> c = np.array([[0, 1], [2, 3], [4, 5], [6, 7]])
>>> c[b]
array([[2, 3], [6, 7]])
```

The second example is an illustration of using integer masks on arrays:

```
>>> a = np.array([[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
 [13, 14, 15, 16]])
>>> a[[2, 1]]
array([[9, 10, 11, 12], [5, 6, 7, 8]])
>>> a[[-2, -1]]          # select rows from the end
array([[ 9, 10, 11, 12], [13, 14, 15, 16]])
>>> a[[2, 3], [0, 1]]    # take elements at (2, 0) and (3, 1)
array([9, 14])
```

Note

The mask array must have the same length as the axis that it's indexing.

Numerical operations on arrays

We are getting familiar with creating and accessing ndarrays. Now, we continue to the next step, applying some mathematical operations to array data without writing any for loops, of course, with higher performance.

Scalar operations will propagate the value to each element of the array:

```
>>> a = np.ones(4)
>>> a * 2
array([2., 2., 2., 2.])
>>> a + 3
array([4., 4., 4., 4.])
```

All arithmetic operations between arrays apply the operation element wise:

```
>>> a = np.ones([2, 4])
>>> a * a
array([[1., 1., 1., 1.], [1., 1., 1., 1.]])
>>> a + a
array([[2., 2., 2., 2.], [2., 2., 2., 2.]])
```

Also, here are some examples of comparisons and logical operations:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 1, 5, 3])
>>> a == b
array([True, False, False, False], dtype=bool)
```

```
>>> np.array_equal(a, b)      # array-wise comparison
False
```

```
>>> c = np.array([1, 0])
>>> d = np.array([1, 1])
>>> np.logical_and(c, d)     # logical operations
array([True, False])
```

Array functions:

Many helpful array functions are supported in NumPy for analyzing data. We will list some part of them that are common in use. Firstly, the transposing function is another kind of reshaping form that returns a view on the original data array without copying anything:

```
>>> a = np.array([[0, 5, 10], [20, 25, 30]])
>>> a.reshape(3, 2)
array([[0, 5], [10, 20], [25, 30]])
>>> a.T
array([[0, 20], [5, 25], [10, 30]])
```

In general, we have the swapaxes method that takes a pair of axis numbers and returns a view on the data, without making a copy:

```
>>> a = np.array([[[0, 1, 2], [3, 4, 5]],
 [[6, 7, 8], [9, 10, 11]]])
>>> a.swapaxes(1, 2)
array([[[0, 3],
    [1, 4],
    [2, 5]],
   [[6, 9],
    [7, 10],
    [8, 11]]])
```

The transposing function is used to do matrix computations; for example, computing the inner matrix product XT.X using np.dot:

```
>>> a = np.array([[1, 2, 3],[4,5,6]])
>>> np.dot(a.T, a)
array([[17, 22, 27],
   [22, 29, 36],
```

```
 [27, 36, 45]])
```

Sorting data in an array is also an important demand in processing data. Let's take a look at some sorting functions and their use:

```
>>> a = np.array ([[6, 34, 1, 6], [0, 5, 2, -1]])

>>> np.sort(a)     # sort along the last axis
array([[1, 6, 6, 34], [-1, 0, 2, 5]])

>>> np.sort(a, axis=0)    # sort along the first axis
array([[0, 5, 1, -1], [6, 34, 2, 6]])

>>> b = np.argsort(a)    # fancy indexing of sorted array
>>> b
array([[2, 0, 3, 1], [3, 0, 2, 1]])
>>> a[0][b[0]]
array([1, 6, 6, 34])

>>> np.argmax(a)    # get index of maximum element
1
```

See the following table for a listing of array functions:

Function	Description	Example
sin, cos, tan, cosh, sinh, tanh, arcos, arctan, deg2rad	Trigonometric and hyperbolic functions	`>>> a = np.array([0.,30., 45.])` `>>> np.sin(a * np.pi / 180)`

Function	Description	Example
		array([0., 0.5, 0.7071678])
around, round, rint, fix, floor, ceil, trunc	Rounding elements of an array to the given or nearest number	>>> a = np.array([0.34, 1.65]) >>> np.round(a) array([0., 2.])
sqrt, square, exp, expm1, exp2, log, log10, log1p, logaddexp	Computing the exponents and logarithms of an array	>>> np.exp(np.array([2.25, 3.16])) array([9.4877, 23.5705])
add, negative, multiply, devide, power, substract, mod, modf, remainder	Set of arithmetic functions on arrays	>>> a = np.arange(6) >>> x1 = a.reshape(2,3) >>> x2 = np.arange(3) >>> np.multiply(x1, x2) array([[0,1,4],[0,4,10]])

Function	Description	Example
greater, greater_equal, less, less_equal, equal, not_equal	Perform elementwise comparison: >, >=, <, <=, ==, !=	`>>> np.greater(x1, x2) array([[False, False, False], [True, True, True]], dtype = bool)`

Data processing using arrays

With the NumPy package, we can easily solve many kinds of data processing tasks without writing complex loops. It is very helpful for us to control our code as well as the performance of the program. In this part, we want to introduce some mathematical and statistical functions.

See the following table for a listing of mathematical and statistical functions:

Function	Description	Example
sum	Calculate the sum of all the elements in an array or along the axis	`>>> a = np.array([[2,4], [3,5]])` `>>> np.sum(a, axis=0)` `array([5, 9])`
prod	Compute the product of array elements over the given axis	`>>> np.prod(a, axis=1)` `array([8, 15])`
diff	Calculate the discrete difference along the given axis	`>>> np.diff(a, axis=0)` `array([[1,1]])`
gradient	Return the gradient of an array	`>>> np.gradient(a)` `[array([[1., 1.], [1., 1.]]), array([[2., 2.], [2., 2.]])]`
cross	Return the cross product of two arrays	`>>> b = np.array([[1,2], [3,4]])`

Function	Description	Example
		>>> np.cross(a,b) array([0, -3])

std, var	Return standard deviation and variance of arrays	>>> np.std(a)
		1.1180339
		>>> np.var(a)
		1.25

mean	Calculate arithmetic mean of an array	>>> np.mean(a) 3.5
where	Return elements, either from x or y, that satisfy a condition	>>> np.where([[True, True], [False, True]], [[1,2],[3,4]], [[5,6],[7,8]]) array([[1,2], [7, 4]])

unique	Return the sorted unique values in an array	>>> id = np.array(['a', 'b', 'c', 'c', 'd'])
		>>> np.unique(id)
		array(['a', 'b', 'c', 'd'], dtype='|S1')

intersect1d	Compute the sorted and common elements in two arrays	>>> a = np.array(['a', 'b', 'a', 'c', 'd', 'c'])
		>>> b = np.array(['a', 'xyz', 'klm', 'd'])
		>>> np.intersect1d(a,b)
		array(['a', 'd'], dtype='|S3')

Loading and saving data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in NumPy package.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension .npy by the np.save function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.save('test1.npy', a)
```

Note

The library automatically assigns the .npy extension, if we omit it.

If we want to store several arrays into a single file in an uncompressed .npz format, we can use the np.savez function, as shown in the following example:

```
>>> a = np.arange(4)
>>> b = np.arange(7)
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The .npz file is a zipped archive of files named after the variables they contain. When we load an .npz file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz')
>>> dic['arr0']
array([0, 1, 2, 3])
```

Another way to save array data into a file is using the np.savetxt function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
>>> # e.g., set comma as separator between elements
>>> np.savetxt('test3.out', x, delimiter=',')
```

Loading an array

We have two common functions such as np.load and np.loadtxt, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
```

```
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

Loading and saving data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in NumPy package.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension .npy by the np.save function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.save('test1.npy', a)
```

Note

The library automatically assigns the .npy extension, if we omit it.

If we want to store several arrays into a single file in an uncompressed .npz format, we can use the np.savez function, as shown in the following example:

```
>>> a = np.arange(4)
>>> b = np.arange(7)
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The .npz file is a zipped archive of files named after the variables they contain. When we load an .npz file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz')
```

```
>>> dic['arr0']
array([0, 1, 2, 3])
```

Another way to save array data into a file is using the np.savetxt function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
>>> # e.g., set comma as separator between elements
>>> np.savetxt('test3.out', x, delimiter=',')
```

Loading an array

We have two common functions such as np.load and np.loadtxt, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension .npy by the np.save function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.save('test1.npy', a)
```

Note

The library automatically assigns the .npy extension, if we omit it.

If we want to store several arrays into a single file in an uncompressed .npz format, we can use the np.savez function, as shown in the following example:

```
>>> a = np.arange(4)
>>> b = np.arange(7)
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The .npz file is a zipped archive of files named after the variables they contain. When we load an .npz file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz')
>>> dic['arr0']
array([0, 1, 2, 3])
```

Another way to save array data into a file is using the np.savetxt function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
>>> # e.g., set comma as separator between elements
>>> np.savetxt('test3.out', x, delimiter=',')
```

Loading an array

We have two common functions such as np.load and np.loadtxt, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

Loading an array

We have two common functions such as np.load and np.loadtxt, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

Linear algebra with NumPy

Linear algebra is a branch of mathematics concerned with vector spaces and the mappings between those spaces. NumPy has a package called **linalg** that supports powerful linear algebra functions. We can use these functions to find eigenvalues and eigenvectors or to perform singular value decomposition:

```
>>> A = np.array([[1, 4, 6],
    [5, 2, 2],
    [-1, 6, 8]])
>>> w, v = np.linalg.eig(A)
>>> w                    # eigenvalues
array([-0.111 + 1.5756j, -0.111 – 1.5756j, 11.222+0.j])
>>> v                    # eigenvector
array([[-0.0981 + 0.2726j, -0.0981 – 0.2726j, 0.5764+0.j],
    [0.7683+0.j, 0.7683-0.j, 0.4591+0.j],
    [-0.5656 – 0.0762j, -0.5656 + 0.00763j, 0.6759+0.j]])
```

The function is implemented using the geev Lapack routines that compute the eigenvalues and eigenvectors of general square matrices.

Another common problem is solving linear systems such as Ax = b with A as a matrix and x and b as vectors. The problem can be solved easily using the numpy.linalg.solve function:

```
>>> A = np.array([[1, 4, 6], [5, 2, 2], [-1, 6, 8]])
>>> b = np.array([[1], [2], [3]])
>>> x = np.linalg.solve(A, b)
>>> x
array([[-1.77635e-16], [2.5], [-1.5]])
```

The following table will summarise some commonly used functions in the numpy.linalg package:

Function	Description	Example

dot	Calculate the dot product of two arrays	>>> a = np.array([[1, 0],[0, 1]])
		>>> b = np.array([[4, 1],[2, 2]])
		>>> np.dot(a,b)
		array([[4, 1],[2, 2]])

inner, outer	Calculate the inner and outer product of two arrays	>>> a = np.array([1, 1, 1])
		>>> b = np.array([3, 5, 1])
		>>> np.inner(a,b)
		9

linalg.norm	Find a matrix or vector norm	>>> a = np.arange(3)
		>>> np.linalg.norm(a)
		2.23606

linalg.det	Compute the determinant of an array	>>> a = np.array([[1,2],[3,4]])
		>>> np.linalg.det(a)
		-2.0

linalg.inv	Compute the inverse of a matrix	>>> a = np.array([[1,2],[3,4]])

Function	Description	Example
		>>> np.linalg.inv(a) array([[-2., 1.],[1.5, -0.5]])

linalg.qr	Calculate the QR decomposition	>>> a = np.array([[1,2],[3,4]])
		>>> np.linalg.qr(a)
		(array([[0.316, 0.948], [0.948, 0.316]]),
		array([[3.162, 4.427], [0., 0.632]]))

linalg.cond	Compute the condition number of a matrix	>>> a = np.array([[1,3],[2,4]])
		>>> np.linalg.cond(a)
		14.933034

trace	Compute the sum of the diagonal element	>>> np.trace(np.arange(6)). reshape(2,3)) 4

8.Python Programs for creation and manipulation of DataFrames using Pandas Library

An overview of the Pandas package

Pandas is a Python package that supports fast, flexible, and expressive data structures, as well as computing functions for data analysis. The following are some prominent features that Pandas supports:

- Data structure with labeled axes. This makes the program clean and clear and avoids common errors from misaligned data.
- Flexible handling of missing data.
- Intelligent label-based slicing, fancy indexing, and subset creation of large datasets.
- Powerful arithmetic operations and statistical computations on a custom axis via axis label.
- Robust input and output support for loading or saving data from and to files, databases, or HDF5 format.

Related to Pandas installation, we recommend an easy way, that is to install it as a part of Anaconda, a cross-platform distribution for data analysis and scientific computing. You can refer to the reference at http://docs.continuum.io/anaconda/ to download and install the library.

After installation, we can use it like other Python packages. Firstly, we have to import the following packages at the beginning of the program:

```
>>> import pandas as pd

>>> import numpy as np
```

The Pandas data structure

Let's first get acquainted with two of Pandas' primary data structures: the Series and the DataFrame. They can handle the majority of use cases in finance, statistic, social science, and many areas of engineering.

Series

A Series is a one-dimensional object similar to an array, list, or column in table. Each item in a Series is assigned to an entry in an index:

```
>>> s1 = pd.Series(np.random.rand(4),
            index=['a', 'b', 'c', 'd'])
>>> s1
a   0.6122
b   0.98096
c   0.3350
d   0.7221
dtype: float64
```

By default, if no index is passed, it will be created to have values ranging from 0 to N-1, where N is the

length of the Series:

```
>>> s2 = pd.Series(np.random.rand(4))
>>> s2
0   0.6913
1   0.8487
2   0.8627
3   0.7286
dtype: float64
```

We can access the value of a Series by using the index:

```
>>> s1['c']
0.3350
>>>s1['c'] = 3.14
>>> s1['c', 'a', 'b']
c   3.14
a   0.6122
```

```
b    0.98096
```

This accessing method is similar to a Python dictionary. Therefore, Pandas also allows us to initialize a Series object directly from a Python dictionary:

```
>>> s3 = pd.Series({'001': 'Nam', '002': 'Mary',
                '003': 'Peter'})
>>> s3
001    Nam
002    Mary
003    Peter
dtype: object
```

Sometimes, we want to filter or rename the index of a Series created from a Python dictionary. At such times, we can pass the selected index list directly to the initial function, similarly to the process in the above example. Only elements that exist in the index list will be in the Series object. Conversely, indexes that are missing in the dictionary are initialized to default NaN values by Pandas:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                '003': 'Peter'}, index=[
                '002', '001', '024', '065'])
>>> s4
002    Mary
001    Nam
024    NaN
065    NaN
dtype:   object
ect
```

The library also supports functions that detect missing data:

```
>>> pd.isnull(s4)
002    False
001    False
024    True
065    True
dtype: bool
```

Similarly, we can also initialize a Series from a scalar value:

```
>>> s5 = pd.Series(2.71, index=['x', 'y'])
>>> s5
x    2.71
y    2.71
dtype: float64
```

A Series object can be initialized with NumPy objects as well, such as ndarray. Moreover, Pandas can automatically align data indexed in different ways in arithmetic operations:

```
>>> s6 = pd.Series(np.array([2.71, 3.14]), index=['z', 'y'])
>>> s6
z    2.71
y    3.14
dtype: float64
>>> s5 + s6
x    NaN
y    5.85
z    NaN
dtype: float64
```

The DataFrame

The DataFrame is a tabular data structure comprising a set of ordered columns and rows. It can be thought of as a group of Series objects that share an index (the column names). There are a number of ways to initialize a DataFrame object. Firstly, let's take a look at the common example of creating DataFrame from a dictionary of lists:

```
>>> data = {'Year': [2000, 2005, 2010, 2014],
        'Median_Age': [24.2, 26.4, 28.5, 30.3],
        'Density': [244, 256, 268, 279]}
>>> df1 = pd.DataFrame(data)
>>> df1
    Density   Median_Age   Year
0  244        24.2         2000
1  256        26.4         2005
2  268        28.5         2010
3  279        30.3         2014
```

By default, the DataFrame constructor will order the column alphabetically. We can edit the default order by passing the column's attribute to the initializing function:

```
>>> df2 = pd.DataFrame(data, columns=['Year', 'Density',
                                  'Median_Age'])
>>> df2
    Year   Density   Median_Age
0   2000   244       24.2
1   2005   256       26.4
2   2010   268       28.5
3   2014   279       30.3
>>> df2.index
Int64Index([0, 1, 2, 3], dtype='int64')
```

#We can provide the index labels of a DataFrame similar to a Series:

```
>>> df3 = pd.DataFrame(data, columns=['Year', 'Density',
              'Median_Age'], index=['a', 'b', 'c', 'd'])
>>> df3.index
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

We can construct a DataFrame out of nested lists as well:

```
>>> df4 = pd.DataFrame([
   ['Peter', 16, 'pupil', 'TN', 'M', None],
   ['Mary', 21, 'student', 'SG', 'F', None],
   ['Nam', 22, 'student', 'HN', 'M', None],
   ['Mai', 31, 'nurse', 'SG', 'F', None],
   ['John', 28, 'laywer', 'SG', 'M', None]],
columns=['name', 'age', 'career', 'province', 'sex', 'award'])
```

Columns can be accessed by column name as a Series can, either by dictionary-like notation or as an attribute, if the column name is a syntactically valid attribute name:

```
>>> df4.name    # or df4['name']
0   Peter
1   Mary
2   Nam
3   Mai
4   John
Name: name, dtype: object
```

To modify or append a new column to the created DataFrame, we specify the column name and the value we want to assign:

```
>>> df4['award'] = None
>>> df4
   name age   career province  sex award
0  Peter  16    pupil      TN    M  None
1   Mary  21  student      SG    F  None
2   Nam   22  student      HN  M  None
3   Mai    31    nurse      SG    F    None
4   John    28   lawer      SG    M    None
```

Using a couple of methods, rows can be retrieved by position or name:

```
>>> df4.ix[1]
name          Mary
age            21
career     student
province        SG
sex             F
award        None
Name: 1, dtype: object
```

A DataFrame object can also be created from different data structures such as a list of dictionaries, a dictionary of Series, or a record array. The method to initialize a DataFrame object is similar to the examples above.

Another common case is to provide a DataFrame with data from a location such as a text file. In this situation, we use the read_csv function that expects the column separator to be a comma, by default. However, we can change that by using the sep parameter:

```
# person.csv file
name,age,career,province,sex
Peter,16,pupil,TN,M
```

```
Mary,21,student,SG,F
Nam,22,student,HN,M
Mai,31,nurse,SG,F
John,28,lawer,SG,M
# loading person.cvs into a DataFrame
>>> df4 = pd.read_csv('person.csv')
>>> df4
   name  age  career  province  sex
0  Peter  16  pupil     TN      M
1  Mary   21  student   SG      F
2  Nam    22  student   HN      M
3  Mai    31  nurse     SG      F
4  John   28  laywer    SG      M
```

While reading a data file, we sometimes want to skip a line or an invalid value. As for

Pandas 0.16.2, read_csv supports over 50 parameters for controlling the loading process. Some common useful parameters are as follows:

- sep: This is a delimiter between columns. The default is comma symbol.
- dtype: This is a data type for data or columns.
- header: This sets row numbers to use as the column names.
- skiprows: This skips line numbers to skip at the start of the file.
- error_bad_lines: This shows invalid lines (too many fields) that will, by default, cause an exception, such that no DataFrame will be returned. If we set the value of this parameter as false, the bad lines will be skipped.

Moreover, Pandas also has support for reading and writing a DataFrame directly from or to a database such as the read_frame or write_frame function within the Pandas module.

The essential basic functionality

Pandas supports many essential functionalities that are useful to manipulate Pandas data structures. In this book, we will focus on the most important features regarding exploration and analysis.

Reindexing and altering labels

Reindex is a critical method in the Pandas data structures. It confirms whether the new or modified data satisfies a given set of labels along a particular axis of Pandas object.

First, let's view a reindex example on a Series object:

```
>>> s2.reindex([0, 2, 'b', 3])
0    0.6913
2    0.8627
b    NaN
3    0.7286
dtype: float64
```

When reindexed labels do not exist in the data object, a default value of NaN will be automatically assigned to the position; this holds true for the DataFrame case as well:

```
>>> df1.reindex(index=[0, 2, 'b', 3],
        columns=['Density', 'Year', 'Median_Age','C'])
   Density  Year  Median_Age     C
0      244  2000        24.2   NaN
2      268  2010        28.5   NaN
b      NaN   NaN         NaN   NaN
3      279  2014        30.3   NaN
```

We can change the NaN value in the missing index case to a custom value by setting the fill_value parameter. Let us take a look at the arguments that the reindex function supports, as shown in the following table:

Argument	Description
index	This is the new labels/index to conform to.
method	This is the method to use for filling holes in a reindexed object. The default setting is unfill gaps. pad/ffill: fill values forward

Argument	Description
	backfill/bfill: fill values backward nearest: use the nearest value to fill the gap
copy	This return a new object. The default setting is true.
level	The matches index values on the passed multiple index level.
fill_value	This is the value to use for missing values. The default setting is NaN.
limit	This is the maximum size gap to fill in forward or backward method.

Head and tail

In common data analysis situations, our data structure objects contain many columns and a large number of rows. Therefore, we cannot view or load all information of the objects. Pandas supports functions that allow us to inspect a small sample. By default, the functions return five elements, but we can set a custom number as well. The following example shows how to display the first five and the last three rows of a longer Series:

```
>>> s7 = pd.Series(np.random.rand(10000))
>>> s7.head()
0    0.631059
1    0.766085
2    0.066891
3    0.867591
4    0.339678
dtype: float64
>>> s7.tail(3)
9997    0.412178
9998    0.800711
9999    0.438344
dtype: float64
```

We can also use these functions for DataFrame objects in the same way.

Binary operations

Firstly, we will consider arithmetic operations between objects. In different indexes objects case, the expected result will be the union of the index pairs. We will not explain this again because we had an example about it in the above section (s5 + s6). This time, we will show another example with a DataFrame:

```
>>> df5 = pd.DataFrame(np.arange(9).reshape(3,3),0
                    columns=['a','b','c'])
>>> df5
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
>>> df6 = pd.DataFrame(np.arange(8).reshape(2,4),
                    columns=['a','b','c','d'])
>>> df6
   a  b  c  d
0  0  1  2  3
1  4  5  6  7
>>> df5 + df6
    a   b   c   d
0   0   2   4 NaN
1   7   9  11 NaN
2   NaN NaN NaN NaN
```

The mechanisms for returning the result between two kinds of data structure are similar. A problem that we need to consider is the missing data between objects. In this case, if we want to fill with a fixed value, such as 0, we can use the arithmetic functions such as add, sub, div, and mul, and the function's supported parameters such as fill_value:

```
>>> df7 = df5.add(df6, fill_value=0)
```

```
>>> df7
  a  b  c  d
0  0  2  4  3
1  7  9  11  7
2  6  7  8  NaN
```

Next, we will discuss comparison operations between data objects. We have some supported functions such as **equal** (**eq**), **not equal** (**ne**), **greater than** (**gt**), **less than** (**lt**), **less equal** (**le**), and **greater equal** (**ge**). Here is an example:

```
>>> df5.eq(df6)
     a     b     c     d
0  True  True  True  False
1  False  False  False  False
2  False  False  False  False
```

Functional statistics

The supported statistics method of a library is really important in data analysis. To get inside a big data object, we need to know some summarized information such as mean, sum, or quantile. Pandas supports a large number of methods to compute them. Let's consider a simple example of calculating the sum information of df5, which is a DataFrame object:

```
>>> df5.sum()
a    9
b   12
c   15
dtype: int64
```

When we do not specify which axis we want to calculate sum information, by default, the function will calculate on index axis, which is axis 0:

- **Series**: We do not need to specify the axis.
- **DataFrame**: Columns (axis = 1) or index (axis = 0). The default setting is axis 0.

We also have the skipna parameter that allows us to decide whether to exclude missing data or not. By default, it is set as true:

```
>>> df7.sum(skipna=False)
a    13
b    18
c    23
d   NaN
dtype: float64
```

Another function that we want to consider is describe(). It is very convenient for us to summarize most of the statistical information of a data structure such as the Series and DataFrame, as well:

```
>>> df5.describe()
        a    b    c
count  3.0  3.0  3.0
mean   3.0  4.0  5.0
std    3.0  3.0  3.0
min    0.0  1.0  2.0
25%    1.5  2.5  3.5
50%    3.0  4.0  5.0
75%    4.5  5.5  6.5
max    6.0  7.0  8.0
```

We can specify percentiles to include or exclude in the output by using the percentiles parameter; for example, consider the following:

```
>>> df5.describe(percentiles=[0.5, 0.8])
        a    b    c
count  3.0  3.0  3.0
mean   3.0  4.0  5.0
```

```
std    3.0  3.0  3.0

min    0.0  1.0  2.0

50%    3.0  4.0  5.0

80%    4.8  5.8  6.8

max    6.0  7.0  8.0
```

Here, we have a summary table for common supported statistics functions in Pandas:

Function	Description
idxmin(axis), idxmax(axis)	This compute the index labels with the minimum or maximum corresponding values.
value_counts()	This compute the frequency of unique values.
count()	This return the number of non-null values in a data object.
mean(), median(), min(), max()	This return mean, median, minimum, and maximum values of an axis in a data object.
std(), var(), sem()	These return the standard deviation, variance, and standard error of mean.
abs()	This gets the absolute value of a data object.

Function application

Pandas supports function application that allows us to apply some functions supported in other packages such as NumPy or our own functions on data structure objects. Here, we illustrate two examples of these cases, firstly, using apply to execute the std() function, which is the standard deviation calculating function of the NumPy package:

```
>>> df5.apply(np.std, axis=1)    # default: axis=0

0    0.816497

1    0.816497

2    0.816497

dtype: float64
```

Secondly, if we want to apply a formula to a data object, we can also useapply function by following these steps:

1. Define the function or formula that you want to apply on a data object.

2. Call the defined function or formula via apply. In this step, we also need to figure out the axis that we want to apply the calculation to:

```
3. >>> f = lambda x: x.max() – x.min()    # step 1
4. >>> df5.apply(f, axis=1)              # step 2
5. 0    2
6. 1    2
7. 2    2
8. dtype: int64
9. >>> def sigmoid(x):
10.      return 1/(1 + np.exp(x))
11.   >>> df5.apply(sigmoid)
12.        a         b        c
13.   0  0.500000  0.268941  0.119203
14.   1  0.047426  0.017986  0.006693
15.   2  0.002473  0.000911  0.000335
```

Sorting

There are two kinds of sorting method that we are interested in: sorting by row or column index and sorting by data value.

Firstly, we will consider methods for sorting by row and column index. In this case, we have the sort_index () function. We also have axis parameter to set whether the function should sort by row or column. The ascending option with the true or false value will allow us to sort data in ascending or descending order. The default setting for this option is true:

```
>>> df7 = pd.DataFrame(np.arange(12).reshape(3,4),
                   columns=['b', 'd', 'a', 'c'],
                   index=['x', 'y', 'z'])
>>> df7
   b  d  a  c
x  0  1  2  3
y  4  5  6  7
```

```
z  8  9  10  11
>>> df7.sort_index(axis=1)
    a  b   c  d
x   2  0   3  1
y   6  4   7  5
z  10  8  11  9
```

Series has a method order that sorts by value. For NaN values in the object, we can also have a special treatment via the na_position option:

```
>>> s4.order(na_position='first')
024     NaN
065     NaN
002    Mary
001     Nam
dtype: object
>>> s4
002    Mary
001     Nam
024     NaN
065     NaN
dtype: object
```

Besides that, Series also has the sort() function that sorts data by value. However, the function will not return a copy of the sorted data:

```
>>> s4.sort(na_position='first')
>>> s4
024     NaN
065     NaN
002    Mary
```

```
001     Nam
dtype: object
```

If we want to apply sort function to a DataFrame object, we need to figure out which columns or rows will be sorted:

```
>>> df7.sort(['b', 'd'], ascending=False)
   b  d   a   c
z  8  9  10  11
y  4  5   6   7
x  0  1   2   3
```

if we do not want to automatically save the sorting result to the current data object, we can change the setting of the `inplace` parameter to `False`.

9. Write a Python program for the following.

- Simple Line Plots,
- Adjusting the Plot: Line Colors and Styles, Axes Limits, Labeling Plots,
- Simple Scatter Plots,
- Histograms,
- Customizing Plot Legends,
- Choosing Elements for the Legend,
- Boxplot
- Multiple Legends,
- Customizing Colorbars,
- Multiple Subplots,
- Text and Annotation,
- Customizing Ticks

Simple Line Plots and Adjusting the Plot: Line Colors and Styles, Axes Limits, Labeling Plots

The matplotlib API primer

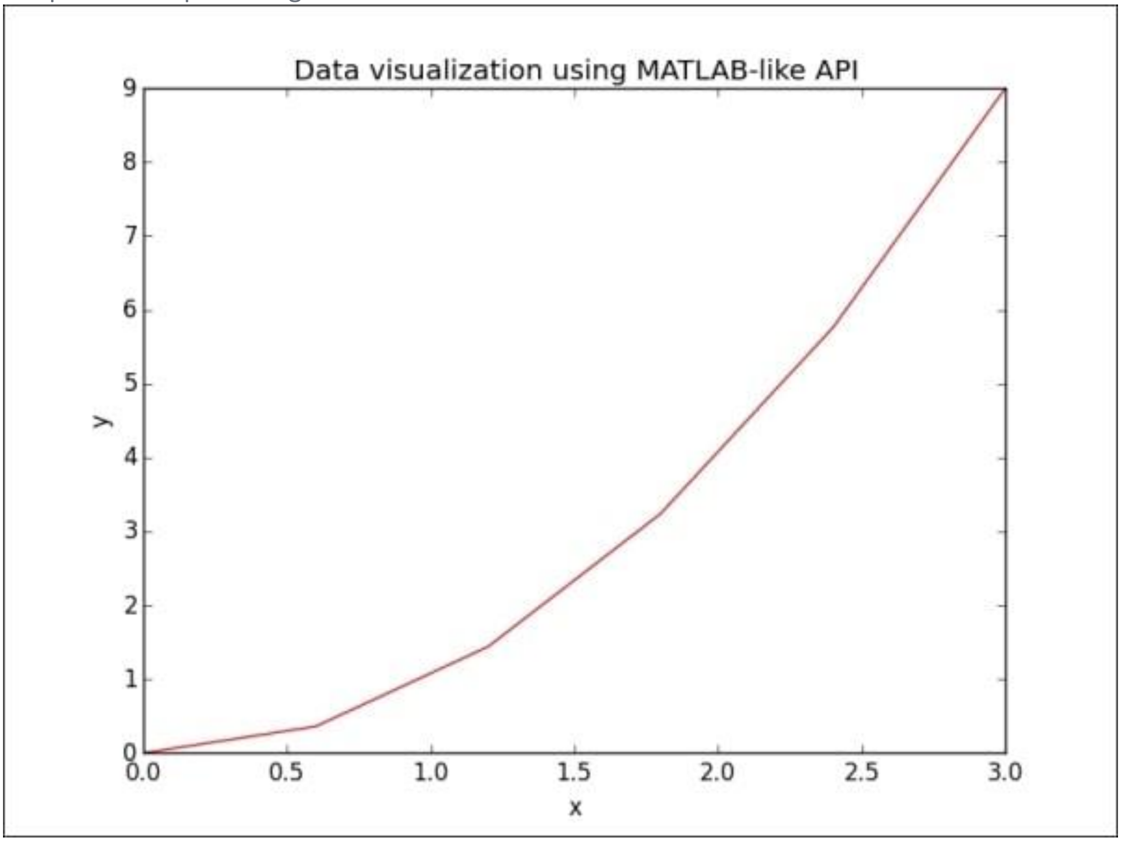
The easiest way to get started with plotting using matplotlib is often by using the MATLAB API that is supported

by the package:

```
>>> import matplotlib.pyplot as plt
>>> from numpy import *
>>> x = linspace(0, 3, 6)
>>> x
array([0., 0.6, 1.2, 1.8, 2.4, 3.])
>>> y = power(x,2)
>>> y
array([0., 0.36, 1.44, 3.24, 5.76, 9.])
>>> figure()
>>> plot(x, y, 'r')
>>> xlabel('x')
```

```
>>> ylabel('y')
>>> title('Data visualization in MATLAB-like API')
>>> plt.show()
```

The output for the preceding command is as follows:



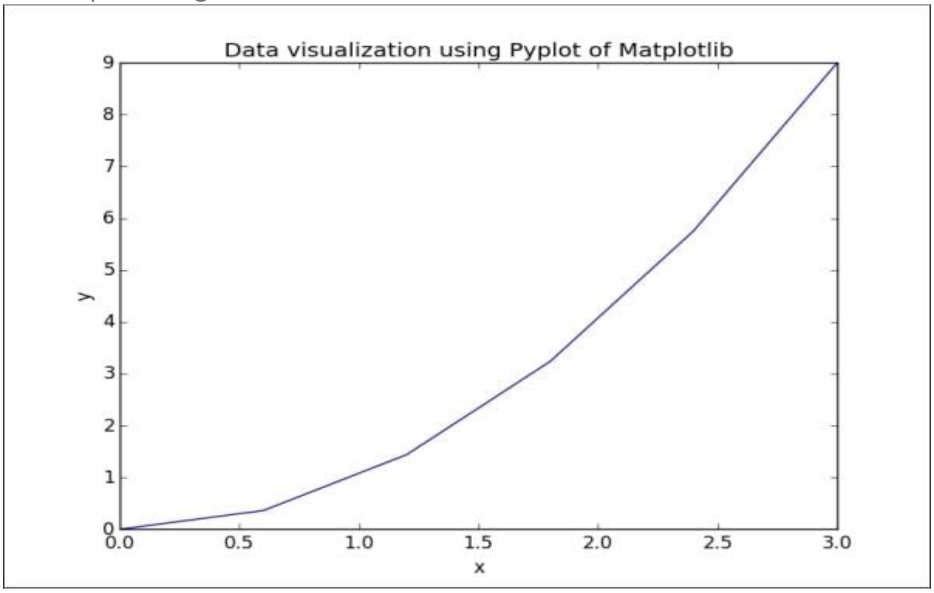
However, star imports should not be used unless there is a good reason for doing so. In the case of matplotlib, we can use the canonical import:

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

The preceding example could then be written as follows:

```
>>> plt.plot(x, y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('Data visualization using Pyplot of Matplotlib')
>>> plt.show()
```

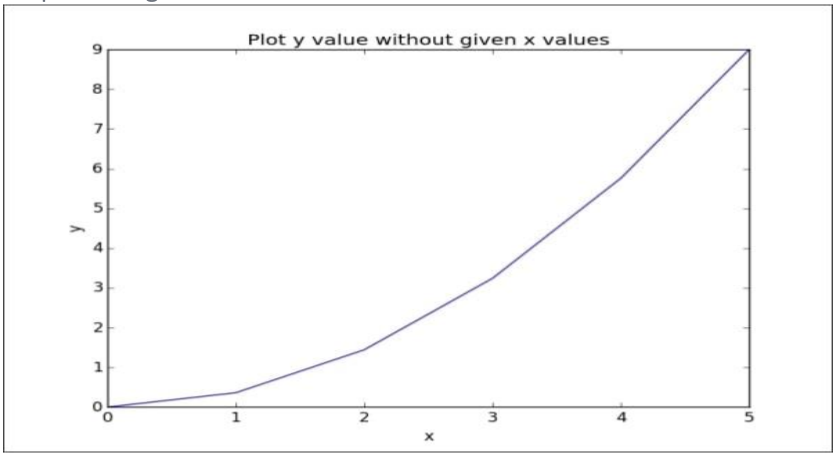
The output for the preceding command is as follows:



If we only provide a single argument to the plot function, it will automatically use it as the y values and generate the x values from 0 to N-1, where N is equal to the number of values:

```
>>> plt.plot(y)

>>> plt.xlabel('x')

>>> plt.ylabel('y')

>>> plt.title('Plot y value without given x values')

>>> plt.show()
```

The output for the preceding command is as follows:



By default, the range of the axes is constrained by the range of the input x and y data. If we want to specify the viewport of the axes, we can use the axis() method to set custom ranges. For example, in the previous visualization, we could increase the range of the x axis from [0, 5] to [0, 6], and that of the y axis from [0, 9] to [0, 10], by writing the following command:

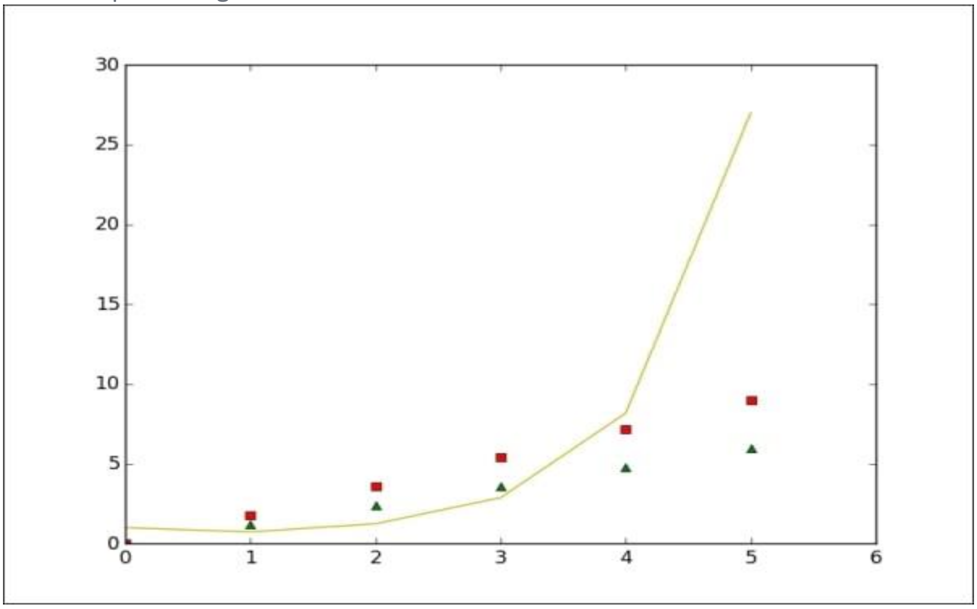
```
>>> plt.axis([0, 6, 0, 12])
```

Line properties

The default line format when we plot data in matplotlib is a solid blue line, which is abbreviated as b-. To change this setting, we only need to add the symbol code, which includes letters as color string and symbols as line style string, to the plot function. Let us consider a plot of several lines with different format styles:

```
>>> plt.plot(x*2, 'g^', x*3, 'rs', x**x, 'y-')
>>> plt.axis([0, 6, 0, 30])
>>> plt.show()
```

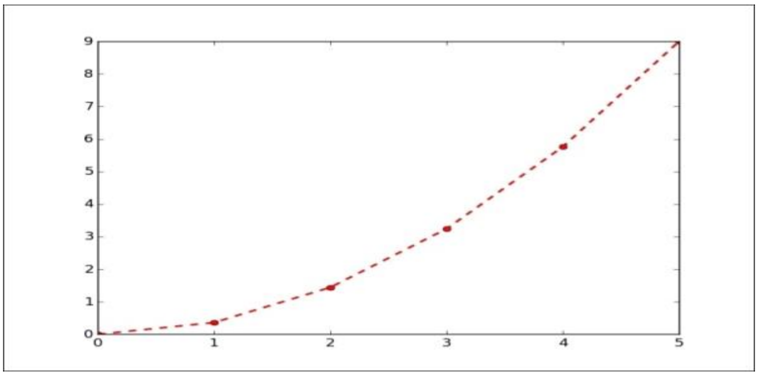
The output for the preceding command is as follows:



There are many line styles and attributes, such as color, line width, and dash style, that we can choose from to control the appearance of our plots. The following example illustrates several ways to set line properties:

```
>>> line = plt.plot(y, color='red', linewidth=2.0)
>>> line.set_linestyle('--')
>>> plt.setp(line, marker='o')
>>> plt.show()
```

The output for the preceding command is as follows:



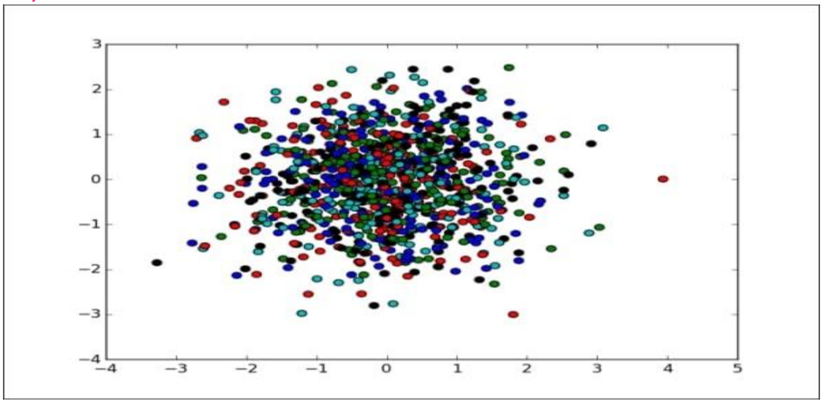
The following table lists some common properties of the line2d plotting:

Property	Value type	Description
color or c	Any matplotlib color	This sets the color of the line in the figure
dashes	On/off	This sets the sequence of ink in the points
data	np.array xdata, np.array ydata	This sets the data used for visualization
linestyle or ls	['-' | '—' |'-.' | ':' | ...]	This sets the line style in the figure
linewidth or lw	Float value in points	This sets the width of line in the figure
marker	Any symbol	This sets the style at data points in the figure

Simple Scatter Plots,

Scatter plots

A scatter plot is used to visualize the relationship between variables measured in the same dataset. It is easy to plot a simple scatter plot, using the plt.scatter() function, that requires numeric columns for both the x and y axis:



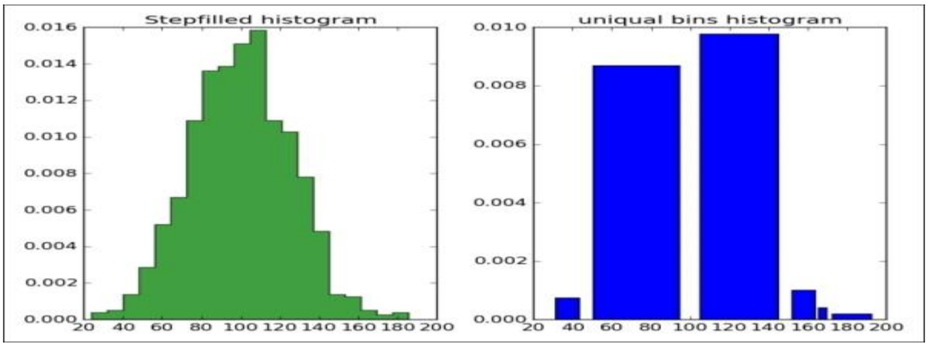
Let's take a look at the command for the preceding output:

```
>>> X = np.random.normal(0, 1, 1000)

>>> Y = np.random.normal(0, 1, 1000)

>>> plt.scatter(X, Y, c = ['b', 'g', 'k', 'r', 'c'])

>>> plt.show()
```

Histograms

Histogram plots

A histogram represents the distribution of numerical data graphically. Usually, the range of values is partitioned into bins of equal size, with the height of each bin corresponding to the frequency of values within that bin:



The command for the preceding output is as follows:

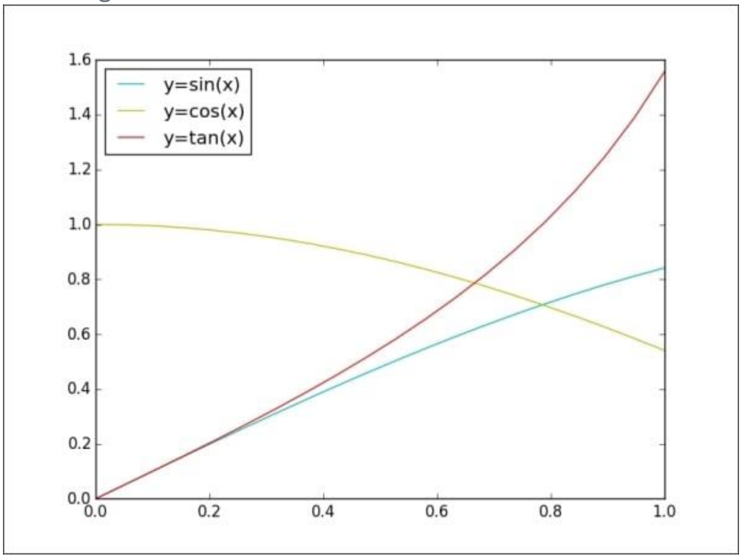
```
>>> mu, sigma = 100, 25
>>> fig, (ax0, ax1) = plt.subplots(ncols=2)
>>> x = mu + sigma * np.random.randn(1000)
>>> ax0.hist(x,20, normed=1, histtype='stepfilled',
          facecolor='g', alpha=0.75)
>>> ax0.set_title('Stepfilled histogram')
>>> ax1.hist(x, bins=[100,150, 165, 170, 195] normed=1,
         histtype='bar', rwidth=0.8)
>>> ax1.set_title('uniquel bins histogram')
>>> # automatically adjust subplot parameters to give specified padding
>>> plt.tight_layout()
>>> plt.show()
```

#Legends and annotations

Legends are an important element that is used to identify the plot elements in a figure. The easiest way to show a legend inside a figure is to use the label argument of the plot function, and show the labels by calling the plt.legend() method:

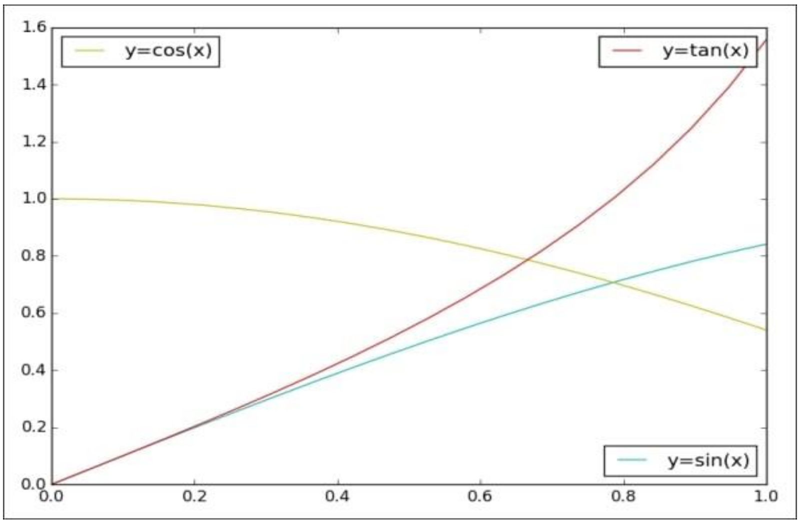
```
>>> x = np.linspace(0, 1, 20)
>>> y1 = np.sin(x)
>>> y2 = np.cos(x)
>>> y3 = np.tan(x)
>>> plt.plot(x, y1, 'c', label='y=sin(x)')
>>> plt.plot(x, y2, 'y', label='y=cos(x)')
>>> plt.plot(x, y3, 'r', label='y=tan(x)')
>>> plt.lengend(loc='upper left')
>>> plt.show()
```

The output for the preceding command as follows:



The loc argument in the legend command is used to figure out the position of the label box. There are several valid location options: lower left, right, upper left, lower center, upper right, center, lower right, upper right, center right, best, upper center, and center left. The default position setting is upper right. However, when we set an invalid location option that does not exist in the above list, the function automatically falls back to the best option.

If we want to split the legend into multiple boxes in a figure, we can manually set our expected labels for plot lines, as shown in the following image:



The output for the preceding command is as follows:

```
>>> p1 = plt.plot(x, y1, 'c', label='y=sin(x)')
>>> p2 = plt.plot(x, y2, 'y', label='y=cos(x)')
>>> p3 = plt.plot(x, y3, 'r', label='y=tan(x)')
>>> lsin = plt.legend(handles=p1, loc='lower right')
>>> lcos = plt.legend(handles=p2, loc='upper left')
>>> ltan = plt.legend(handles=p3, loc='upper right')
>>> # with above code, only 'y=tan(x)' legend appears in the figure
>>> # fix: add lsin, lcos as separate artists to the axes
>>> plt.gca().add_artist(lsin)
>>> plt.gca().add_artist(lcos)
>>> # automatically adjust subplot parameters to specified padding
>>> plt.tight_layout()
>>> plt.show()
```

The other element in a figure that we want to introduce is the annotations which can consist of text, arrows, or other shapes to explain parts of the figure in detail, or to emphasize some special data points. There are different methods for showing annotations, such as text, arrow, and annotation.

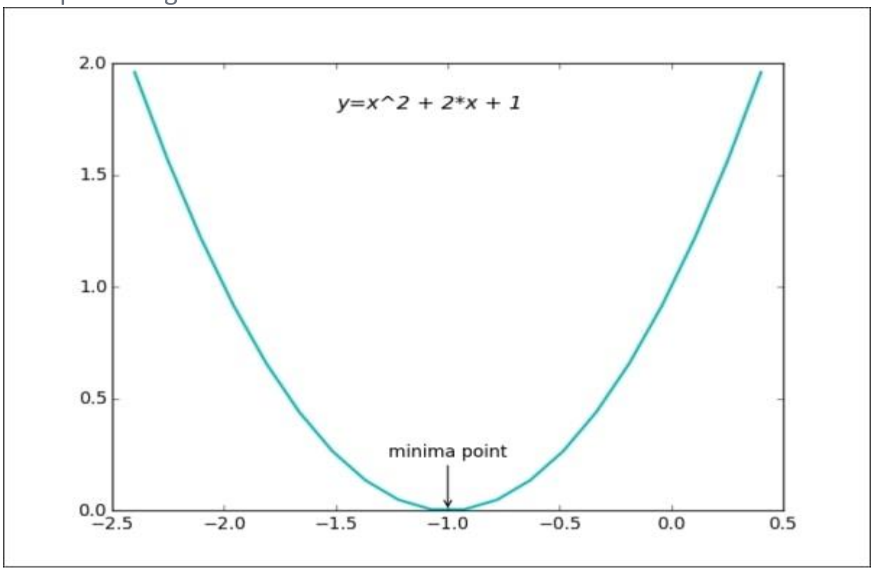
- The text method draws text at the given coordinates (x, y) on the plot; optionally with custom properties. There are some common arguments in the function: x, y, label text, and font-related properties that can be passed in via fontdict, such as family, fontsize, and style.

- The annotate method can draw both text and arrows arranged appropriately. Arguments of this function are s (label text), xy (the position of element to annotation), xytext (the position of the label s), xycoords (the string that indicates what type of coordinate xy is), and arrowprops (the dictionary of line properties for the arrow that connects the annotation).

Here is a simple example to illustrate the annotate and text functions:

```
>>> x = np.linspace(-2.4, 0.4, 20)
>>> y = x*x + 2*x + 1
>>> plt.plot(x, y, 'c', linewidth=2.0)
>>> plt.text(-1.5, 1.8, 'y=x^2 + 2*x + 1',
          fontsize=14, style='italic')
>>> plt.annotate('minima point', xy=(-1, 0),
            xytext=(-1, 0.3),
            horizontalalignment='center',
            verticalalignment='top',
            arrowprops=dict(arrowstyle='->',
            connectionstyle='arc3'))
>>> plt.show()
```

The output for the preceding command is as follows:

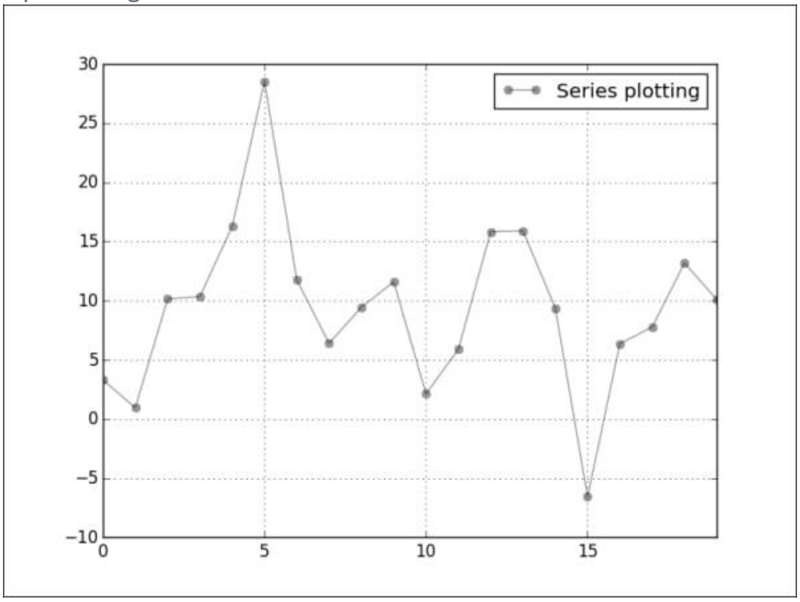


Plotting functions with Pandas

For Series or DataFrame objects in Pandas, most plotting types are supported, such as line, bar, box, histogram, and scatter plots, and pie charts. To select a plot type, we use the kind argument of the plot function. With no kind of plot specified, the plot function will generate a line style visualization by default , as in the following example:

```
>>> s = pd.Series(np.random.normal(10, 8, 20))
>>> s.plot(style='ko—', alpha=0.4, label='Series plotting')
>>> plt.legend()
>>> plt.show()
```

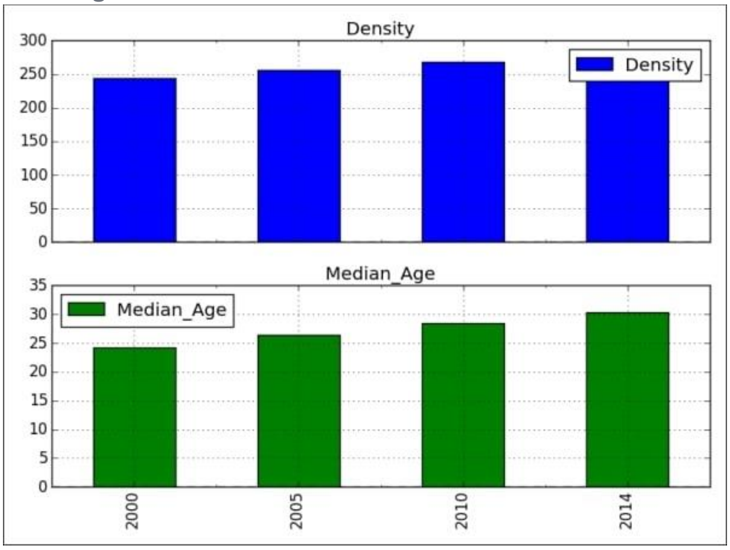
The output for the preceding command is as follows:



Another example will visualize the data of a DataFrame object consisting of multiple columns:

```
>>> data = {'Median_Age': [24.2, 26.4, 28.5, 30.3],
       'Density': [244, 256, 268, 279]}
>>> index_label = ['2000', '2005', '2010', '2014'];
>>> df1 = pd.DataFrame(data, index=index_label)
>>> df1.plot(kind='bar', subplots=True, sharex=True)
>>> plt.tight_layout();
>>> plt.show()
```

The output for the preceding command is as follows:



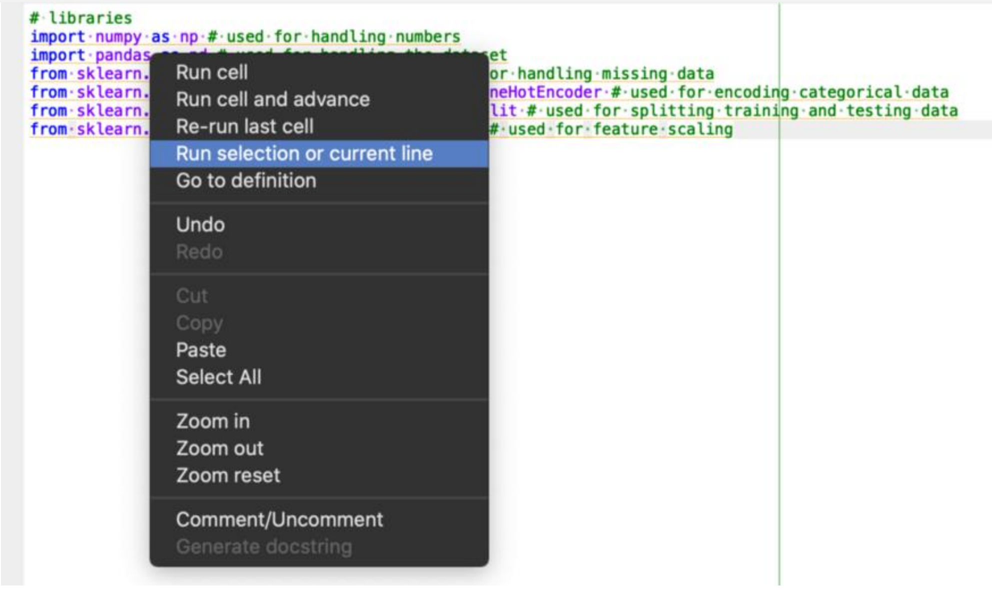
The plot method of the DataFrame has a number of options that allow us to handle the plotting of the columns. For example, in the above DataFrame visualization, we chose to plot the columns in separate subplots. The following table lists more options:

Argument	Value	Description
subplots	True/False	The plots each data column in a separate subplot
logy	True/False	The gets a log-scale y axis
secondary_y	True/False	The plots data on a secondary y axis
sharex, sharey	True/False	The shares the same x or y axis, linking sticks and limits

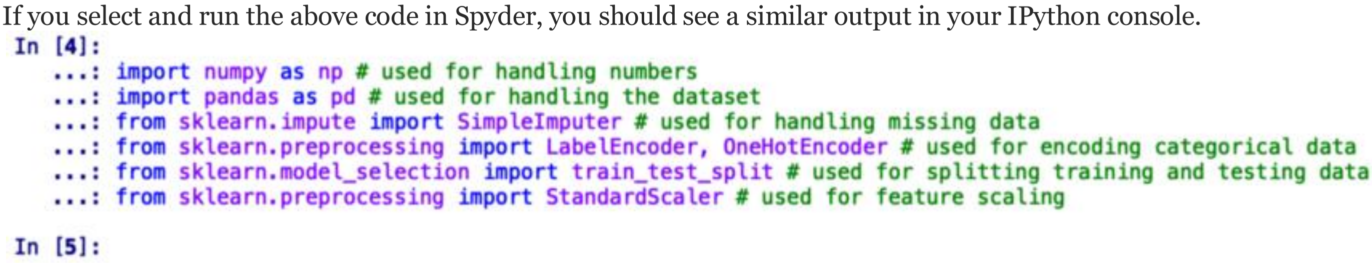
10. Python Programs for Data preprocessing: Handling missing values, handling categorical data, bringing features to same scale, selecting meaningful features

Importing the libraries:

```
# libraries
import numpy as np # used for handling numbers
import pandas as pd # used for handling the dataset
from sklearn.impute import SimpleImputer # used for handling missing datafrom sklearn.preprocessing
import LabelEncoder, OneHotEncoder # used for encoding categorical datafrom sklearn.model_selection
import train_test_split # used for splitting training and testing datafrom sklearn.preprocessing
import StandardScaler # used for feature scaling
```



If you select and run the above code in Spyder, you should see a similar output in your IPython console.


In [4]:
...: import numpy as np # used for handling numbers
...: import pandas as pd # used for handling the dataset
...: from sklearn.impute import SimpleImputer # used for handling missing data
...: from sklearn.preprocessing import LabelEncoder, OneHotEncoder # used for encoding categorical data
...: from sklearn.model_selection import train_test_split # used for splitting training and testing data
...: from sklearn.preprocessing import StandardScaler # used for feature scaling

In [5]:

If you see any import errors, try to install those packages explicitly using `pip` command as follows.

```
pip install <package-name>
```

Importing the Dataset

First of all, let us have a look at the dataset we are going to use for this particular example. You can find the https://github.com/tarunlnmiit/machine_learning/blob/master/DataPreprocessing.csv

1	Region	Age	Income	Online Shopper
2	India	49	86400	No
3	Brazil	32	57600	Yes
4	USA	35	64800	No
5	Brazil	43	73200	No
6	USA	45		Yes
7	India	40	69600	Yes
8	Brazil		62400	No
9	India	53	94800	Yes
10	USA	55	99600	No
11	India	42	80400	Yes

In order to import this dataset into our script, we are apparently going to use pandas as follows.

```
dataset = pd.read_csv('Data.csv') # to import the dataset into a variable# Splitting the attributes into independent and dependent attributes

X = dataset.iloc[:, :-1].values # attributes to determine dependent variable / Class
Y = dataset.iloc[:, -1].values # dependent variable / Class
```

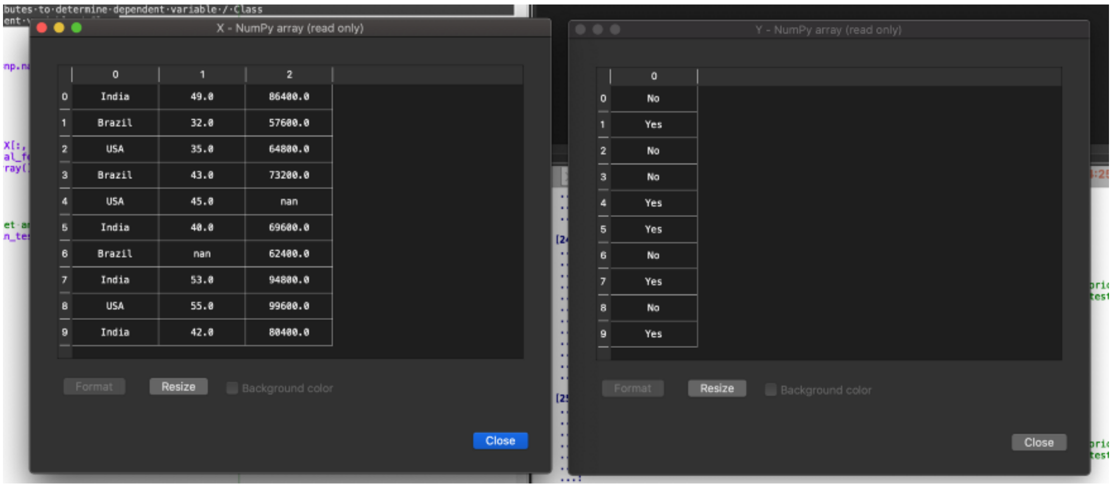
When you run this code section, you should not see any errors, if you do make sure the script and the *Data.csv* are in the same folder. When successfully executed, you can move to variable explorer in the Spyder UI and you will see the following three variables.

Name	Type	Size	Value
X	object	(10, 3)	ndarray object of numpy module
Y	object	(10,)	Min: 'No' Max: 'Yes'
dataset	DataFrame	(10, 4)	Column names: Region, Age, Income, Online Shopper

When you double click on each of these variables, you should see something similar.

dataset - DataFrame

Index	Region	Age	Income	Online Shopper
0	India	49	86400	No
1	Brazil	32	57600	Yes
2	USA	35	64800	No
3	Brazil	43	73200	No
4	USA	45	nan	Yes
5	India	40	69600	Yes
6	Brazil	nan	62400	No
7	India	53	94800	Yes
8	USA	55	99600	No
9	India	42	80400	Yes



If you face any errors in order to see these data variables, try to upgrade Spyder to Spyder version 4.

Handling of Missing Data

Well the first idea is to remove the lines in the observations where there is some missing data. But that can be quite dangerous because imagine this data set contains crucial information. It would be quite dangerous to remove an observation. So we need to figure out a better idea to handle this problem. And another idea that’s actually the most common idea to handle missing data is to take the mean of the columns.

If you noticed in our dataset, we have two values missing, one for age column in 7th data row and for Income column in 5th data row. Missing values should be handled during the data analysis. So, we do that as follows.

```
# handling the missing data and replace missing values with nan from numpy and replace with mean of all the other values

imputer = SimpleImputer(missing_values=np.nan, strategy='mean') imputer = imputer.fit(X[:, 1:])
X[:, 1:] = imputer.transform(X[:, 1:])
```

After execution of this code, the independent variable *X* will transform into the following.

	0	1	2
0	India	49.0	86400.0
1	Brazil	32.0	57600.0
2	USA	35.0	64800.0
3	Brazil	43.0	73200.0
4	USA	45.0	76533.333333…
5	India	40.0	69600.0
6	Brazil	43.777777777…	62400.0
7	India	53.0	94800.0
8	USA	55.0	99600.0
9	India	42.0	80400.0

Here you can see, that the missing values have been replaced by the average values of the respective columns.

Handling of Categorical Data

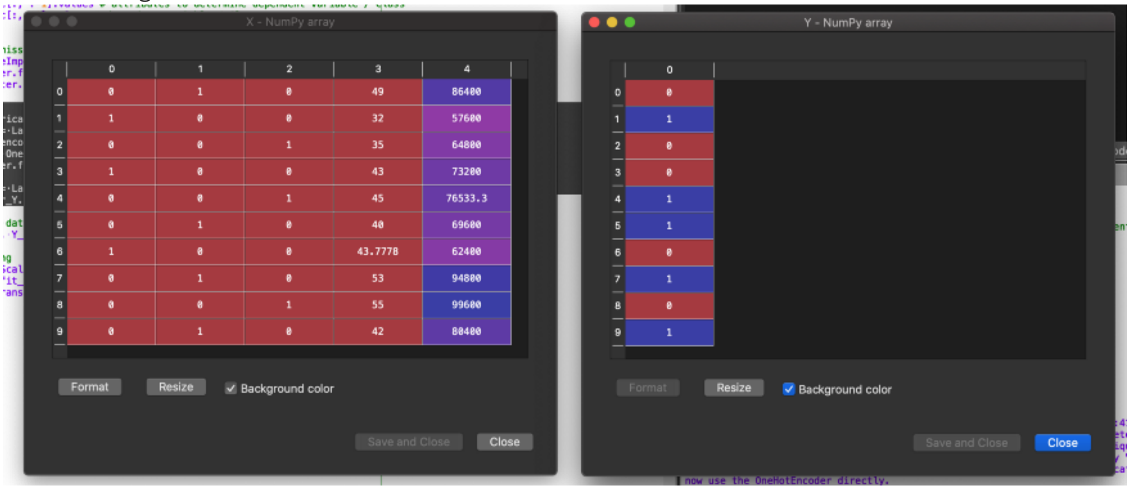
In this dataset we can see that we have two categorical variables. We have the Region variable and the Online Shopper variable. These two variables are categorical variables because simply they contain categories. The Region contains three categories. It's ***India, USA & Brazil*** and the online shopper variable contains two categories. ***Yes*** and ***No*** that's why they're called categorical variables.

You can guess that since machine learning models are based on mathematical equations you can intuitively understand that it would cause some problem if we keep the text here in the categorical variables in the equations because we would only want numbers in the equations. So that's why we need to encode the categorical variables. That is to encode the text that we have here into numbers. To do this we use the following code snippet.

```
# encode categorical data

from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
onehotencoder = OneHotEncoder(categorical_features=[0])
```

```
X = onehotencoder.fit_transform(X).toarray()labelencoder_Y = LabelEncoder()
Y = labelencoder_Y.fit_transform(Y)
```

After execution of this code, the independent variable *X* and dependent variable *Y* will transform into the following.



Here, you can see that the Region variable is now made up of a 3 bit binary variable. The left most bit represents ***India,*** 2nd bit represents ***Brazil*** and the last bit represents ***USA.*** If the bit is **1** then it represents data for that country otherwise not. For *Online Shopper* variable, **1** represents **Yes** and **0** represents **No.**

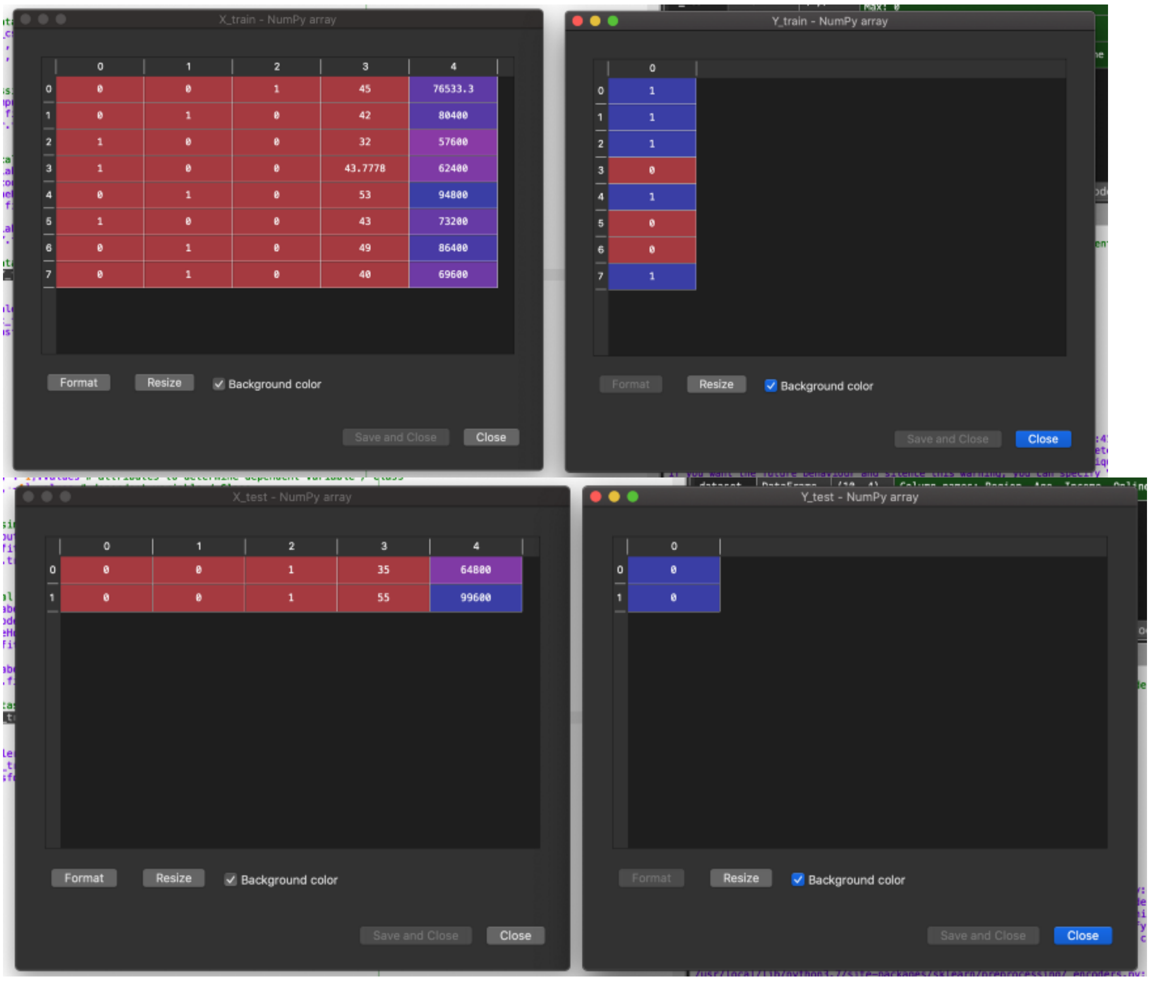
Splitting the dataset into training and testing datasets

Any machine learning algorithm needs to be tested for accuracy. In order to do that, we divide our data set into two parts: **training set** and **testing set.** As the name itself suggests, we use the training set to make the algorithm learn the behaviours present in the data and check the correctness of the algorithm by testing on testing set. In Python, we do that as follows:

```
# splitting the dataset into training set and test set
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=0)
```

Here, we are taking training set to be 80% of the original data set and testing set to be 20% of the original data set. This is usually the ratio in which they are split. But, you can come across sometimes to a 70–30% or 75–25% ratio split. But, you don't want to split it 50–50%. This can lead to ***Model Overfitting.*** This topic is too huge to be covered in the same post. I will cover it in some future post. For now, we are going to split it in 80–20% ratio.

After split, our training set and testing set look like this.



Feature Scaling

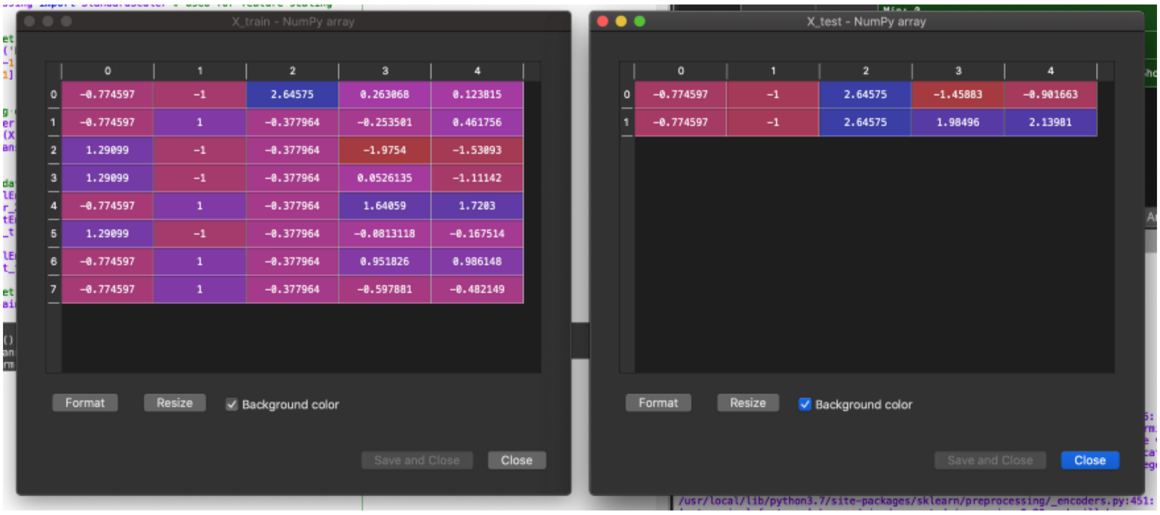
As you can see we have these two columns age and income that contains numerical numbers. You notice that the variables are not on the same scale because the age are going from 32 to 55 and the salaries going from 57.6 K to like 99.6 K.

So because this age variable in the salary variable don't have the same scale. This will cause some issues in your machinery models. And why is that. It's because your machine models a lot of machinery models are based on what is called the Euclidean distance.

We use feature scaling to convert different scales to a standard scale to make it easier for Machine Learning algorithms. We do this in Python as follows:

```
# feature scaling

sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

After the execution of this code, our training independent variable *X* and our testing independent variable *X* and look like this.



This data is now ready to be fed to a Machine Learning Algorithm.

This concludes this post on Data Preprocessing in Python.