Sub setting Lists

The first element has index 0, the second element has index 1. This is called ZERO indexing. We use the index in the square brackets to find the element in the list.

The last element of the list can also be called by using '-1'. This is called Negative Indexing.

Slicing

selecting more than one elements using ':' colon.

[start : end] = start is included. End is excluded. Index specified after the colon is not included.

[:4] = Telling python to start from Index 0.

[5:] = all elements from index 5 to last element of the list.

Manipulating Lists

- Change list elements
- Add list elements
- Remove list elements

Use the list name[index] = new value/new string

The index element gets replaced.

list name [0:2] = [new elements of the list]. New elements are added.

Add elements to the list : list name + new element

del(list name[index])

using list 1 = list 2

y = x

Any change in y is reflected in x.

y = list(x)

y = x[:]

Any change in y does not change x.

Functions

- Function is a piece of reusable code.
- Solves a particular task

- Call function instead of writing code yourself.

ex ; round, max,

round(number, ndigits) : ndigits specifics the no. of decimal places. ndigits is optional.

getting to know the a function :

help()

or

function name

Strings, lists, variables are objects. Python objects also come with a bunch of "Methods". Methods are function that belong to objects.

Example : str has capitalize(), replace().

Index, count is a method. It comes after the variable denoted after a '.'.

Everything is an object in Python and each object have methods associated, depending on the type.

So, a list doesn't have replace.

Methods can change the objects. Some methods don't.

For importing or calling the package we use 'import'

pi is accessed through math.pi

Calling specific function from a package

'from math import radians'

Numpy

We can do basic calculations on list.

Numpy - Numeric Python

Numpy has array called Numpy array

Calculations over entire arrays.

array formed by using 'np.array'.

Now, we can perform operations on Numpy.

Numpy array contains only one type. So, if contains only string or float or integer.

List + List = includes both the list in a single list

array + array = adds elements according to their indices.

You can call array elements using their index.

you can use conditions to check the array. The output is a Boolean.

bmi > 23 - gives Boolean output

bmi[bmi > 23] - gives the elements for which the corresponding Boolean value is True

np.array(list) converts the lists into array. So, we can perform calculations.

2D Numpy arrays

type of array is given as numpy.ndarray, which implies it as numpy n-dimensional array.

We can include two list in an array. Each sublist in the list corresponds to a **row** in a two dimensional numpy array.

np_2d.shape - gives the shape or the no of rows and columns.

Here shape is an attribute not a method. Method has brackets in front of it. All array have same data type. If you change one float to a string, all the array elements will be converted to strings.

To get an element from 2-D array, we call a row then column index.

np_2d [0] [2] OR np_2d [0,2]

zero is index of row. Two is index of column.

np_2d [: , 1:3]

we want all the rows. and only 1st and 2nd columns.

Multiply original array's each column with a Number. Create an array with the values. Multiply that array with the original array.

np.mean() or x.mean() - for a particular row, column or entire array. Where x is the array.

np.median() or x.median() - for a particular row, column or entire array

x.max( ) -

x.min( ) -

np.corrcoef() - Correlation

np.std() - standard deviation

np.sum() -

np.sort() -

this can be done in regular python. But the speed of numpy is more.

np.random.normal() - np.random.normal(distribution mean, distribution std. deviation, no. of samples)

np.column_stack() - specify the columns in the bracket.

np.arange(15) - Create random 15 variable array

np.arange(15).reshape(3,5) - Here reshape (m,n) creates m x n matrix of 15 values. m is the no. of rows (lists) and n is the no. of columns (elements in each list).

You can write array within an array.

b = np.array (np.arange(0,5), 5 + np.arange(0,5) , 10 + np.arange(0,5))

```
c **=** np**.**array([[1, 2], [3, 4]], dtype**=**complex)
```

You can specify the data type using dtype.

Various attributes in Numpy

```
a.ndim:  dimension of array
a.shape:  rows x columns
a.size: no. of elements
a.dtype:  int32 - element type
```

```
a.itemsize:  4 - size of element
a.data: memory location of array
```

np.empty((4,4)) - create matrix of infinitesimal small values.

np.linspace(0,1,10) - 10 values between 0 and 1

n D array

print(np.**arange(24)**.**reshape(2, 3, 4))**

It makes two 3 x 4 matrix.

print(np.**arange(24)**.**reshape(2, 3, 2, 2))**

It makes three 2 x 2 matrix. This process is done 2 times.

```
# Element-wise product
A* B
# matrix product
A@ B
A.dot(B)
```

'*=' and '+=' modify existing arrays instead of creating new arrays.

A *= 3 - Each element is multiplied by 3

Universal functions

- np.add ( ) -
- np.exp ( ) - exponential of each component of A
- np.log ( ) - logarithm of each component of A
- np.sqrt ( )
- np.square

Indexing, Slicing and Iterating

So, in a n-dimensional array how to slice the arrays.

The 1st entry in square bracket is for the matrix to be selected. Other conditions come for the rows and columns.

'c' is (2, 2, 3) 3-d array.

```
# extract all elements
c[::]
c[0:,:]
# extract first elements (2x3 matrix) on first axis
c[0:1,:]
```

0 :1 decides the element-matrix to be selected.

: - implies all the elements

```
# extract second elements (2x3 matrix) on first axis
c[1:,:]
```

1 : - all elements from index 1 are selected.

```
# In both the matrices, fetch 2nd row 3rd & 2nd elements
c[:,1,[2,1]]
```

: - implies all elements are selected.

1 : - 2nd row

[2,1] : - 3rd and 2nd column of the matrix

Field Names and Indexing

Create Field names for each element in an array. Call the elements by using field names instead of index.

Repeat, Tile, Reshape, Transpose, Logical Operations

Duplicate numpy arrays using repeat and tile.

Repeat for duplicating elements

tile creates a matrix of required shape

np.repeat(array_name, no. of times) - `np**.**repeat(x,2)`

np.tile(array_name, (rows,columns)) - `np**.**tile(x(2,1))`

Reshape is transforming and changing rows and columns.

`x **=** np**.**arange(8)`

np.reshape(array_name, (rows, columns) `np**.**reshape(x,(2,4))` or `x**.**reshape(4,2))`

`x**.**reshape(2,4)**.**shape` - Output is the shape of 'x'

Transpose -

`x.reshape(2,4).T` OR

`x.reshape(2,4).transpose()`

Logical -

```
np**.**where(x**>**3)
```

All the elements where x>3 are considered.

Stacking numpy arrays :

- *hstack* for horizontal stacking
- *vstack* for vertical stacking
- *c_* for column-wise stacking
- *r_* for row-wise stacking

```
# Create numpy arrays
x= np.arange(7)
y= np.square(x)
# hstack Horizontal stacking
np.hstack([x,y])
```

x and y stacked in one row.

```
# vstack Vertical stacking
np.vstack([x,y])
```

x an y stacked in two rows

```
# c_ Column-wise stacking
np.c_[x,y]
```

Single column and multiple rows. Elements having same index stacked in the same row.

```
# r_ Row-wise stacking
np.r_[x,y]
```

Single row. All elements stacked in one row.

numpy, pandas, pandas_datareader, datetime