Numpy

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np

a = np.array([100, 200, 300])   #Rank 1 array
print(a)
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a.ndim)             # Prints 1

print(a[0], a[1], a[2])   # Prints "100 200 300"

a[0] = 500                  # Change an element of the array
print(a)                  # Prints "[500, 200, 300]"

b = np.array(
              [
                  [1,2,3],
                  [4,5,6]
              ]
            )
# Create a rank 2 array
print(b.shape)   # Prints "(2, 3)"
print (b.ndim)   # Prints 2
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
[100 200 300]
<class 'numpy.ndarray'>
(3,)
1
100 200 300
[500 200 300]
(2, 3)
2
1 2 4
```

```
# 3 dimensional array

import numpy as np
b = np.array( [
                  [
                      [1,2,3],
                      [4,5,6]
```

```
                    ],

                    [
                        [1,2,3],
                        [4,5,6]
                    ]
                ]
            )

print(b.shape)   # Prints "(2, 3)"
print (b.ndim)

print (b[0,1,2])
```

```
(2, 2, 3)
3
6
```

https://www.quora.com/In-Python-NumPy-what-is-a-dimension-and-axis

```
# Numpy Matrix Operations

import numpy as np


#   1 2      5 6      6  8
#         +        =
#   3 4      7 8      10 12

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))


#   1 2      5 6      -4  -4
#         -        =
#   3 4      7 8      -4  -4


# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))


#   1 2      5 6       5  12
#         *        =
#   3 4      7 8      21  32
```

```


# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]

#   1 2      5 6       0.2  0.3333
#         /        =
#   3 4      7 8       0.428  0.5

print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

[Google SpreadSheet Link]

```
import numpy as np

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

v = np.array([9,10])
w = np.array([11, 12])


# 9 10  dot product   11 12
```

```
# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a,'\n')              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b,'\n')              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c,'\n')               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d,'\n')              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e,'\n')                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

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

[[1. 1.]]

[[7 7]
 [7 7]]

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

[[0.37258962 0.1633431 ]
 [0.83065705 0.64296041]]
```

```
# Similar to Python lists, numpy arrays can be sliced. Since arrays may be mult
# you must specify a slice for each dimension of the array:
```

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array(
              [
                [1,2,3,4],
                [5,6,7,8],
                [9,10,11,12]
              ]
            )

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

print (b,'\n')


x = a[:,:1]
print(x)

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1],'\n')   # Prints "77"

print (b,'\n')

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

```
[[2 3]
```

```
# One can also mix integer indexing with slice indexing.
# However, doing so will yield an array of lower rank than the original array
```

```
# However, doing so will yield an array of lower rank than the original array


import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

```

# Integer array indexing: When you index into numpy arrays using slicing,
# the resulting array view will always be a subarray of the original array.
# In contrast, integer array indexing allows you to construct arbitrary arrays u
# Here is an example:

import numpy as np

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



# When integers are used for indexing. Each element of first dimension is paire

# An example of integer array indexing.
# The returned array will have shape (3,) and

# This is equivalent to [a[0, 0], a[1, 1], a[2, 0]]))
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"




# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
```

```
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

```

```

```
#Boolean array indexing: Boolean array indexing lets you pick out arbitrary eler
#Frequently this type of indexing is used to select the elements of an array tha


import numpy as np

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

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

```
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
[3 4 5 6]
```

```
#One useful trick with integer array indexing is selecting or mutating one elem

import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"
```

```

print(a.shape)
# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
# (0,1,2,3) ([0, 2, 0, 1])
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

```
#Numpy provides many useful functions for performing computations on arrays; on

import numpy as np


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


# 1,2
# 3,4

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
# Python Program illustrating
# numpy.reshape() method

import numpy as np

array = np.arange(8)
print("Original array : \n", array)

# shape array with 2 rows and 2 columns
array = np.arange(4).reshape(2, 2)
print("\narray reshaped with 2 rows and 2 columns : \n", array)

# shape array with 4 rows and 2 columns
array = np.arange(8).reshape(4, 2)
print("\narray reshaped with 4 rows and 2 columns : \n", array)


# shape array with 2 rows and 4 columns
array = np.arange(8).reshape(4 ,2)
print("\narray reshaped with 4 rows and 2 columns : \n", array)

```

```
# Constructs 3D array
array = np.arange(8).reshape(2, 2, 2)
print("\nOriginal array reshaped to 3D : \n", array)

```

```
Original array :
 [0 1 2 3 4 5 6 7]

array reshaped with 2 rows and 2 columns :
 [[0 1]
 [2 3]]

array reshaped with 4 rows and 2 columns :
 [[0 1]
 [2 3]
 [4 5]
 [6 7]]

array reshaped with 4 rows and 2 columns :
 [[0 1]
 [2 3]
 [4 5]
 [6 7]]

Original array reshaped to 3D :
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

```
from numpy import array
# list of data
data = [[11, 22],
    [33, 44],
    [55, 66]]
# array of data
data = array(data)
print('Rows: %d' % data.shape[0])
print('Cols: %d' % data.shape[1])
```

```

# reshape 1D array to 2D Array
from numpy import array
from numpy import reshape
# define array
data = array([11, 22, 33, 44, 55])
print(data.shape)
# reshape
print (data.shape[0])
data = data.reshape((data.shape[0], 1))
print(data.shape)

print (data)
```

```
# A tensor that contains only one number is called a scalar (or scalar tensor,
```

```
# In Numpy, a float32 or float64 number is a scalar tensor (or scalar array).

import numpy as np
x = np.array(12)
print (x.ndim)
```

```
#An array of numbers is called a vector, or 1D tensor.
# A 1D tensor is said to have exactly one axis. Following is a Numpy vector:

a = np.array([100, 200, 300])   #Rank 1 array
print(a.ndim)             # Prints 1
```

```
# Matrices (2D tensors)
# An array of vectors is a matrix, or 2D tensor.
# A matrix has two axes (often referred to rows and columns).
# You can visually interpret a matrix as a rectangular grid of numbers. This is

x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])

print (x.ndim)
```

```
# If you pack such matrices in a new array, you obtain a 3D tensor,
# which you can visually interpret as a cube of numbers. Following is a Numpy 3

x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])

print (x)

print (x.ndim)
```

```
# In deep learning, you’ll generally manipulate tensors that are 0D to 4D
```

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print(train_images.ndim)
print(train_images.shape)
print(train_images.dtype)

#So what we have here is a 3D tensor of 8-bit integers.
#More precisely, it’s an array of 60,000 matrices of 28 × 28 integers.
#Each such matrix is a grayscale image, with coefficients between 0 and 255.
```

Real-world examples of data tensors

Vector data— 2D tensors of shape (samples, features)

Timeseries data or sequence data— 3D tensors of shape (samples, timesteps, features)

Images— 4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)

Video— 5D tensors of shape (samples, frames, height, width, channels)

```
# The simplest example of this type of operation is transposing a matrix;
# to transpose a matrix, simply use the T attribute of an array object:


import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix.

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

print (y)

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
```

```
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

```

import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"

#The line y = x + v works even though x has shape (4, 3)
#and v has shape (3,) due to broadcasting; this line works as if v actually had
#where each row was a copy of v, and the sum was performed elementwise.
```

```

```

Pandas

A pandas series is similar to numpy arrays or lists but with more functionality.

```
# Refer for more
# https://docs.google.com/spreadsheets/d/1SbKUKZIEs9ibZM18y4b3YoWQJW8XMIpbyIaZR

#import numpy as np
import pandas as pd

#a = np.array([1,2,3,4])
series = pd.Series([1,2,3,4])


print (series.describe(),'\n')  # These functions are not available in numpy ar
```

```
count    4.000000
mean     2.500000
std      1.290994
min      1.000000
```

```
min      1.000000
25%      1.750000
50%      2.500000
75%      3.250000
max      4.000000
dtype: float64
```

```
import pandas as pd
series = pd.Series([1,2,3,4])

print (series)
print ("\n")


print (series[0],'\n')  # One can access using the same way like numpy array
print ("\n")


print (series[:2],'\n')  # same slice notation.
```

```
series = pd.Series([1,2,3,4])
for d in series:
  print (d)
```

```
series = pd.Series([1,2,3,4])
print ('\n')

print ('mean', series.mean())
print ('std',  series.std())
print ('max', series.max())
```

```
# Vectorized operations and index arrays
a = pd.Series([1, 2, 3, 4])
b = pd.Series([1, 2, 1, 2])

print (a + b)
print (a * 2)
print (a >= 3)
print (a[a >= 3])
```

```
0    2
1    4
2    4
```

```
3    6
dtype: int64
0    2
1    4
2    6
3    8
dtype: int64
0    False
1    False
2     True
3     True
```

```
# Panda Series Index


population = pd.Series([1415045928,1354051854,326766748])
print (population,'\n')

population = pd.Series([1415045928,1354051854,326766748], index = ["China", "In

print (population,'\n')

# Numpy arrays are like superman version of list
# A Panda Series is like a mix of list and dictionary

print ('***Population[0] =', population [0])
print ('***Population[\'India\']=',population['India'] )
```

```
0    1415045928
1    1354051854
2     326766748
dtype: int64

China    1415045928
India    1354051854
US        326766748
dtype: int64

***Population[0] = 1415045928
***Population['India']= 1354051854
```

```
# How we will do the same thing in numpy

import numpy as np


population = np.array([1415045928,1354051854,326766748])
index = np.array(["China", "India", "US"])

print (population,'\n')

# Numpy arrays are like superman version of list
# A Panda Series is like a mix of list and dictionary


print ('Population of {} is {}'.format(index[1], population[1]))
```

```python
# If no indexes are specified Pandas creates Index
import pandas as pd

numseries = pd.Series([200,400,800])
print (numseries)
```

```python
# iloc and loc

import pandas as pd

population = pd.Series([1415045928,1354051854,326766748], index = ["China", "In

print (population)

print (population[0]) # Accessing using number index without iloc


print (population.iloc[0]) # Accessing using number index

print (population.loc['China']) # Accessing using index
```

```
China    1415045928
India    1354051854
US        326766748
dtype: int64
1415045928
1415045928
1415045928
```

```python
# Searching For a value in Pandas Series



import pandas as pd

population = pd.Series([1415045928,1354051854,326766748], index = ["China", "In

print(population==1415045928,'\n')

value = population[population==1415045928]

print (value,'\n')


print (type(value),'\n')
```

```
China     True
India    False
US       False
dtype: bool

China    1415045928
dtype: int64

<class 'pandas.core.series.Series'>
```

```

import pandas as pd

population = [1415045928,1354051854,326766748]

for i in population:
    if i == 1415045928
value = population[population==1415045928]

print (value)
```

```
import pandas as pd

population = pd.Series([1415045928,1354051854,326766748], index = ["China", "In

value = population[(population==1415045928)| (population==1354051854) ]

print (value)

```

```


# Finding out maximum population
import pandas as pd


print (pd.__version__)

population = pd.Series(data =[1415045928,1354051854,326766748], index = ["China


print ("Country Index with maximum population =",population.values.argmax())
print ("The max population is of country = {} and population ={}".format(popula
                                                                           popula

```

```
0.22.0
Country with maximum population = 0
The max population is of country = 0 and population =1415045928
```

```
import pandas as pd


# Addition when indexes are the same
s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([100, 200, 300, 400], index=['a', 'b', 'c', 'd'])
print (s1 + s2)
```

```
# Indexes have same elements in a different order

s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([100, 200, 300, 400], index=['b', 'd', 'a', 'c'])
print (s1 + s2)
```

```
# Indexes overlap, but do not have exactly the same elements

s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([100, 200, 300, 400], index=['c', 'd', 'e', 'f'])
print (s1 + s2)
```

```
# Indexes do not overlap

s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([10, 20, 30, 40], index=['e', 'f', 'g', 'h'])
print (s1 + s2)
```

```
# Using dropna

s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
```

```
s2 = pd.Series([10, 20, 30, 40], index=['a', 'b', 'g', 'h'])
result  =  s1 + s2
print (result)
print (result.dropna())
```

```
# Using fill_value

s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([10, 20, 30, 40], index=['a', 'b', 'g', 'h'])
result  =  s1.add(s2,fill_value=0)
print (result.dropna())
```

```

# Using Pandas Apply
import pandas as pd

def make_capital(str):
  return str.capitalize()


s1 = pd.Series(['india', 'china', 'brazil'], index=['a', 'b', 'c'])
s2 = s1.apply(make_capital)

print (s2)
```

```
# Using Lambda

s1 = pd.Series(['india', 'china', 'brazil'], index=['a', 'b', 'c'])
s2 = s1.apply(lambda x: x.capitalize())

print (s2)
```

```
# Plotting values
population = pd.Series(data = [1415045928,1354051854,326766748], index = ["China
population.plot()
```

```
# Pandas DataFrame

import pandas as pd

country_df = pd.DataFrame({
    'country':['India','China', 'USA'],
    'population':[1415045928,1354051854,326766748],
    'population2':[1415045928,1354051854,326766748],
    'capital':['Delhi','Bejing','Washington']

})

print (country_df,'\n')

print (country_df.mean())

```

```
# Pandas DataFrame

import pandas as pd

country_df = pd.DataFrame({
    'population':[1415045928,1354051854,326766748],
    'capital':['Delhi','Bejing','Washington'],
    'gdp':[2848231,14092514,20412870]

```

```
},
index = ['India','China', 'USA'],

)

print (country_df,'\n')
print (country_df.loc['India'],'\n')

print (country_df.iloc[0])

```

```
# Pandas DataFrame

import pandas as pd

country_df = pd.DataFrame({
    'population':[1415045928,1354051854,326766748],
    'capital':['Delhi','Bejing','Washington'],
    'gdp':[2848231,14092514,20412870]

},
index = ['India','China', 'USA'],

)

print (country_df)
print (country_df.loc['India','gdp'],'\n')
print (country_df.iloc[0,1],'\n')


print (country_df.loc[ ['India','China'],:])
```

```
# Pandas DataFrame

import pandas as pd

country_df = pd.DataFrame({
    'population':[1415045928,1354051854,326766748],
    'capital':['Delhi','Bejing','Washington'],
    'gdp':[2848231,14092514,20412870]

},
index = ['India','China', 'USA'],

)
print (country_df)

print ('\n')
print (country_df['gdp'],'\n')


```

```
# Accessing Pandas as range of values


#gdp':[2848231,14092514,20412870]

import pandas as pd

country_df = pd.DataFrame({
    'population':[1415045928,1354051854,326766748],
    'capital':['Delhi','Bejing','Washington'],
    'gdp':[2848231,20412870,20412870]


},
index = ['India','China', 'USA'],
)

#print (country_df)


print ('\n------*0*-----\n')
```

```
columns_data = country_df.columns[:]
print (columns_data)

print ('\n------*1*-----\n')
print (country_df[columns_data])


col_data = country_df[columns_data]

print ('\n------*2*------\n')

print (col_data.iloc[:])


print (col_data.loc[:,['capital','gdp']] )  # show two columns using slice

print (col_data.iloc[:,[0,1]] )  # show two columns using iLoc



gdp_df = col_data.loc[:,['gdp']]

print ("\n****The Country with maximum GDP  is ",gdp_df.idxmax())
print ("\n****The maximum GDP  is ",gdp_df.max())
print ("\n****The maximum GDP is ",gdp_df.max().values)


# But there are two maximums

print (country_df[country_df['gdp'] == country_df['gdp'].max()])

print('*****************')
x = country_df[country_df['gdp'] == country_df['gdp'].max()]
print(x.index.values)
```

```python
# Import modules
import pandas as pd
import numpy as np


# Create a dataframe
raw_data = {'first_name': ['Jason', 'Molly', np.nan, np.nan, np.nan],
        'country': ['USA', 'USA', 'France', 'UK', 'UK'],
        'age': [42, 52, 36, 24, 70]}
df = pd.DataFrame(raw_data, columns = ['first_name', 'country', 'age'])
print(df)


# Create variable with TRUE if nationality is USA
american = df['country'] == "USA"

# Create variable with TRUE if age is greater than 50
elderly = df['age'] > 50

# Select all cases where nationality is USA and age is greater than 50
print(df[american & elderly])


```

```
import pandas as pd
#PandasDataForCSV.csv also is presenent in Google SpreadSheet.
#df = pd.read_csv('PandasDataForCSV.csv', index_col='first_name')
url = "http://datasciencemastery.in/wp-content/uploads/2018/10/PandasDataForCSV
df = pd.read_csv(url, index_col='first_name')
print ('\n****** All DataFrame ****')
print(df)
#print all people with age >40
# Print all people greater than age 40
print ('\n****** age>40 ****')
x= df[df['age']>40]
print (x)# prints all columns nationality  age  salary
# Printing the index name
print ('\n******Index Name****')
print(x.index.values)
# Print all people age greater than age 40 and India
print ('\n******age greater than age 40 and India****')
print (df[(df['age']>40) & (df['nationality']=='India')])
# Print all people age greater than age 40 and India
print ('\n@@@******age greater than age 40 and India****')
a = df['age']>40
b = df['nationality']=='India'
print (df[a & b])
# Print maximum age
print ('\n******maximum age****')
# get the row of max value
x = df.loc[df['age'].idxmax()]
print ('\n******Pandas Type ****')
```

```
print (type(x))   # This is pandas series
print ('\n******Pandas All Values ****')
print (x)  # Print all values
print ('\n******Pandas Nationality ****')
print (x.loc['nationality'])  # Print all nationality
print ('\n******Pandas Index or Name of the person with maximum age ****')
print (df['age'].idxmax())


print ('\n******Sort DataFrame ****')
x = df.sort_values("age")
print(x)


print ('\n******InPlace Sorting DataFrame ****')
df.sort_values("age",inplace=True)
print(df)

print ('\n******Sort DataFrame in ascending values ****')
x = df.sort_values("age", ascending =False)
print(x)


print ('\n******Pandas GroupBy ****')
x = df.groupby(['nationality']).groups
print(x)
print ('\n******Pandas GroupBy -2****')
for name,group in df.groupby(['nationality']):
    print (name, group)

```

```
import matplotlib.pyplot as plt

# *****Getting mean age grouped by country and plotting it
```

```
# *****Getting mean age grouped by country and plotting it
x = df.groupby(['nationality'])['age'].mean()

plt.bar(x.index,x)
plt.xlabel('Country')
plt.ylabel('Age')
#plt.xticks(np.arange(12))
plt.grid(True)
plt.title('Age by nationality')

print(x)




print ('\n******Pandas GroupBy ****')
x = df.groupby(['nationality']).groups
print(x)
print ('\n******Pandas GroupBy -2****')
for name,group in df.groupby(['nationality']):
    print (name)
    print (group)

```

```
# Pandas DataFrame output as numpy values

import pandas as pd

country_df = pd.DataFrame({
    'population':[1415045928,1354051854,326766748],
    'capital':['Delhi','Bejing','Washington'],
    'gdp':[2848231,14092514,20412870]

},
index = ['India','China', 'USA'],

)

print (country_df.values) # output as numpy values

```

```
# Pandas axis

import pandas as pd

df = pd.DataFrame({'A': [0, 1, 2], 'B': [3, 4, 5]})

print (df,'\n')

print (df.sum(),'\n')

print (df.sum(axis=0),'\n')
print (df.sum(axis=1),'\n')
```

```
# Vector operations for data frames
import pandas as pd

#Examples of vectorized operations on DataFrames:
# Adding DataFrames with the column names

df1 = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
df2 = pd.DataFrame({'a': [10, 20, 30], 'b': [40, 50, 60], 'c': [70, 80, 90]})
print (df1 + df2,'\n')

# Adding DataFrames with overlapping column names
df1 = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
```

```
df1 = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
df2 = pd.DataFrame({'d': [10, 20, 30], 'c': [40, 50, 60], 'b': [70, 80, 90]})
print (df1 + df2,'\n')

# Adding DataFrames with overlapping row indexes
df1 = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]},
                   index=['row1', 'row2', 'row3'])
df2 = pd.DataFrame({'a': [10, 20, 30], 'b': [40, 50, 60], 'c': [70, 80, 90]},
                   index=['row4', 'row3', 'row2'])
print (df1 + df2,'\n')
```

```
#applymap() applies a function to every single element in the entire dataframe.



import pandas as pd


# DataFrame applymap()

df = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [10, 20, 30],
    'c': [5, 10, 15]
})

def add_one(x):
    return x + 1

print (df.applymap(add_one))

```

```
import numpy as np
import pandas as pd

df = pd.DataFrame({
    'a': [4, 5, 3, 1, 2],
    'b': [20, 10, 40, 50, 30],
    'c': [25, 20, 5, 15, 10]
})

# Change False to True for this block of code to see what it does

# DataFrame apply() - use case 2
if False:
    print df.apply(np.mean)
    print df.apply(np.max)

def second_largest(df):
    '''
    Fill in this function to return the second-largest value of each
    column of the input DataFrame.
    '''
    return None
```

```python
#Adding a DataFrame to a Series

import pandas as pd


# Adding a Series to a square DataFrame

s = pd.Series([1, 2, 3, 4])

print (s,'\n')
df = pd.DataFrame({
    0: [10, 20, 30, 40],
    1: [50, 60, 70, 80],
    2: [90, 100, 110, 120],
    3: [130, 140, 150, 160]
})

print (df)
print ('') # Create a blank line between outputs
print (df + s)
```

```python

# Adding a Series to a one-row DataFrame

s = pd.Series([1, 2, 3, 4])
print (s,'\n')

df = pd.DataFrame({0: [10], 1: [20], 2: [30], 3: [40]})

print (df)
print ('') # Create a blank line between outputs
print (df + s)
```

```python
# Adding a Series to a one-column DataFrame
s = pd.Series([1, 2, 3, 4])
df = pd.DataFrame({0: [10, 20, 30, 40]})

print (df)
print ('') # Create a blank line between outputs
print (df + s)
```

```python

# Adding when DataFrame column names match Series index
```

```
s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
df = pd.DataFrame({
    'a': [10, 20, 30, 40],
    'b': [50, 60, 70, 80],
    'c': [90, 100, 110, 120],
    'd': [130, 140, 150, 160]
})

print (df)
print ('') # Create a blank line between outputs
print (df + s)


```

```
# Adding when DataFrame column names don't match Series index

s = pd.Series([1, 2, 3, 4])
df = pd.DataFrame({
    'a': [10, 20, 30, 40],
    'b': [50, 60, 70, 80],
    'c': [90, 100, 110, 120],
    'd': [130, 140, 150, 160]
})

print (df)
print ('') # Create a blank line between outputs
print (df + s)
```

```
# Reading from a CSV.

import pandas as pd

df = pd.read_csv('https://raw.githubusercontent.com/datasciencemastery/pandas-nu
print(df,'\n')


print (df[df.itemcode=='MAG'],'\n')


print ((df.itemcode=='MAG')& (df.amount>100),'\n')
print (df[(df.itemcode=='MAG')& (df.amount>100)] )

```

```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) = boston_ho(a+)using.loa
print (train_data.shape)
print (test_data.shape)
```

Numpy Broadcasting

https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html

```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])

print (a+b)
```

```
import numpy as np

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


print (a+5)  # Think a+5 as above..
```

```
M = np.ones((3, 3))
print(M)
print('\n')

print(M + a)
```

```
#https://docs.google.com/spreadsheets/d/1rtBUAu9m6kJ6fQNHdVt8X0CAQQ9-jkMf8N3E4a


M = np.ones((2, 3))
print(M)

print('\n')

a = np.arange(3)
print(a)

print('\n')
print(M + a)
```

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


[0 1 2]


[[1. 2. 3.]
 [1. 2. 3.]]
```

```
a = np.arange(3).reshape((3, 1))
print(a,'\n')

b = np.arange(3)
print(b,'\n')


print(a + b)


```

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

[0 1 2]

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

```
a =nparange(12).reshape(3,4)
print(a)
```