Python Packages
Module 1: Pandas DataFrame

Learning objectives

1. Data structures to process the data.
2. Reading files and data operations.
3. Merge with different files.

Data processing is an important part of analyzing the data, because the data is not always available in desired format.Various processing is required before analyzing the data such as cleaning, restructuring or

merging etc. Numpy, Scipy, Cython and Panda are the tools available in python which can be used for fast processing of the data. Further, Pandas are built on the top of Numpy.

Pandas provides a rich set of functions to process various types of data. Further, working with Panda is fast, easy and more expressive than other tools. Pandas provides fast data processing as Numpy along with flexible data manipulation techniques as spreadsheets and relational databases. Lastly, pandas integrates well with matplotlib library, which makes it a very handy tool for analyzing the data.

Data structures

Pandas provides two very useful data structures to process the data i.e. Series and DataFrame.

Series

The Series is a one-dimensional array that can store various data types, including mix data types. The row labels in a Series are called the index. Any list, tuple and dictionary can be converted into Series using 'series' method.

```
import pandas as pd                          #converting tuple to Series
h = ('AA','2019/06/18',100,10.2)
s  = pd.Series(h)
type (s)
print(s)

0          AA
1     2019/06/18
2          100
3          10.2
dtype: object

d = {'name': 'OKG', 'Date':'2018-10-20', 'shares': '100'}
ds = pd.Series(d)                            #converting dict to Series

print(ds)

name            OKG
Date         2018-10-20
shares          100
dtype: object
```

Note that in the tuple-conversion, the indexes are set to '0, 1, 2 and 3'. We can provide custom index names as follows.

```
f = ['FB','1998-09-03', 90,4.5]
f = pd.Series(f, index = ['name', 'date', 'shares','price'])
print(f)

name            FB
```

```
date        1998-09-03
shares          90
price           4.5
dtype: object

sh= f['shares']
print(sh)
90

f[0]
'FB'
```

Elements of the Series can be accessed using index name e.g. f[‘shares’] or f[0] in below code. Further, specific elements can be selected by providing the index in the list,

```
sh= f[['shares', 'price']]
print(sh)

shares      90
price       4.5
dtype: object
```

DataFrame

DataFrame is the widely used data structure of pandas. Note that, Series are used to work with one dimensional array, whereas DataFrame can be used with two dimensional arrays. DataFrame has two different indexes i.e. column-index and row-index.

The most common way to create a DataFrame is by using the dictionary of equal-length list as shown below.Further, all the spreadsheets and text files are read as DataFrame, therefore it is very important data structure of pandas.

```
data = {'name': ['AA','IBM','GOOG'],
      'date': ['2001-12-01','2012-02-10','2010-04-09'],
      'shares': [100, 30, 90],
      'price': [12.3, 10.3, 32.2]
      }
df = pd.DataFrame(data)
print(df)

    name     date   shares  price
0    AA  2001-12-01  100   12.3
1   IBM  2012-02-10  30    10.3
2  GOOG  2010-04-09  90    32.2
```

Additional columns can be added after defining a DataFrame as below,

```
df['owner'] = 'Unknown'
print(df)
```

```
    name    date    shares  price owner
0    AA  2001-12-01  100   12.3   Unknown
1   IBM  2012-02-10  30    10.3   Unknown
2  GOOG  2010-04-09  90    32.2   Unknown
```

Currently, the row index are set to 0, 1 and 2. These can be changed using '`index`' attribute as below,

```
df.index = ['one','two','three']
print(df)
```

```
        name        date      shares  price    owner
one       AA    2001-12-01      100   12.3     Unknown
two       IBM    2012-02-10     30    10.3     Unknown
three    GOOG    2010-04-09     90    32.2     Unknown
```

Further, any column of the DataFrame can be set as index using '`set_index()`' attribute, as shown below,

```
df = df.set_index(['name'])
print(df)
```

```
          date    shares  price  owner
name
AA    2001-12-01      100   12.3  Unknown
IBM    2012-02-10     30   10.3  Unknown
GOOG  2010-04-09      90   32.2  Unknown
```

Data can be accessed in two ways i.e. using row and column index,

```
M = df['shares']
print(M)
```

```
name
AA    100
IBM   30
GOOG  90
Name: shares, dtype: int64
```

```
q = df.ix['AA']
print(q)
```

```
date       2001-12-01
shares          100
price           12.3
owner        Unknown
```

Any column can be deleted using '`del`' or '`drop`' commands,

```
del df['owner']
df
        date         shares  price
name
AA    2001-12-01      100   12.3
IBM   2012-02-10      30   10.3
GOOG  2010-04-09      90   32.2

n=df.drop('shares', axis = 1)
print(n)
         date  price
name
AA    2001-12-01   12.3
IBM   2012-02-10   10.3
GOOG  2010-04-09   32.2
```

Reading files

Two data files are used i.e. '`titles.csv`' and '`cast.csv`'. The '`titles.csv`' file contains a list of movies with the releasing year; whereas '`cast.csv`' file has five columns which store the title of movie, releasing year, star-casts, type(actor/actress), characters and ratings for actors.

```
import pandas as pd
casts = pd.read_csv('cast.csv', index_col=None)
sh = casts.head()
print(sh)
```

```
              title     year   name     type       character              n
0     Closet Monster    2015  Buffy #1 actor       Buffy 4               31.0
1     Suuri illusioni   1985   Homo $   actor       Guests               22.0
2   Battle of the Sexes 2017  $hutter  actor        Bobby Riggs Fan      10.0
3  Secret in Their Eyes 2015  $hutter  actor      2002 Dodger Fan        NaN
4       Steve Jobs      2015  $hutter  actor      1988 Opera House Patron NaN
```

```
t= pd.read_csv('titles.csv', index_col =None)
mh = t.tail()
```

```
print(mh)

                title                 year
49995          Rebel                 1970
49996          Suzanne               1996
49997          Bomba                 2013
49998       Aao Jao Ghar Tumhara     1984
49999        Mrs. Munck              1995
```

- `read_csv`: read the data from the csv file.
- `index_col` = None: there is no index i.e. first column is data
- `head()`: show only first five elements of the DataFrame
- `tail()`: show only last five elements of the DataFrame
- `len`: 'len' command can be used to see the total number of rows in the file

Data operations

Row and column selection

Any row or column of the DataFrame can be selected by passing the name of the column or rows. After selecting one from DataFrame, it becomes one-dimensional therefore it is considered as a Series.

- `ix:` use '`ix`' command to select a row from the DataFrame

```
t = titles['title']
t.head()

0 The Rising Son
1 The Thousand Plane Raid
2 Crucea de piatra
3 Country
4 Gaiking II
Name: title, dtype: object

titles.ix[0]
title The Rising Son
year 1990
Name: 0, dtype: object
```

Filter Data

Data can be filtered by providing some boolean expressions in DataFrame. For example, in the below code, movies which released after 1985 are filtered out from the DataFrame '`titles`' and stored in a new DataFrame i.e. after85.

```
after85 = titles[titles['year'] > 1985]
```

```
after85.head()
```

```
        title                year
0 The Rising Son             1990
2 Crucea de piatra           1993
3 Country                    2000
4 Gaiking II                 2011
5 Medusa (IV)                2015
```

Note: When we pass the boolean results to DataFrame, then panda will show all the results which correspond to True (rather than displaying True and False).

```
t = titles
movies90 = t[ (t['year']>=1990) & (t['year']<2000) ]
movies90.head()
```

```
         Title               year
0 The Rising Son             1990
2 Crucea de piatra           1993
12 Poka Makorer Ghar Bosoti  1996
19 Maa Durga Shakti          1999
24 Conflict of Interest      1993
```

Sorting

Sorting can be performed using '`sort_index`' or '`sort_values`' keywords

```
t = titles
macbeth = t[ t['title'] == 'Macbeth']
macbeth.head()
```

```
     title          year
4226 Macbeth        1913
9322 Macbeth        2006
11722 Macbeth       2013
17166 Macbeth       1997
25847 Macbeth       1998
```

Note that in above filtering operation, the data is sorted by index i.e. by default '`sort_index`' operation is used as shown below,

```
macbeth = t[ t['title'] == 'Macbeth'].sort_index()
macbeth.head()
```

```
       title             year
```

```
4226 Macbeth            1913
9322 Macbeth            2006
11722 Macbeth           2013
17166 Macbeth           1997
25847 Macbeth           1998
```

To sort the data by values, the '`sort_value`' option can be used. In below code, data is sorted by year now,

```
macbeth = t[ t['title'] == 'Macbeth'].sort_values('year')
macbeth.head()
```

```
      title  year
4226 Macbeth 1913
17166 Macbeth 1997
25847 Macbeth 1998
9322 Macbeth 2006
11722 Macbeth 2013
```

Null values

Note that, various columns may contains no values, which are usually filled as `NaN`.

```
casts.ix[3:4]
```

```
      title             year     name     type    character  n
3 Secret in Their Eyes  2015 $hutter actor 2002 Dodger Fan NaN
4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron    NaN
```

These null values can be easily selected, unselected or contents can be replaced by any other values e.g. empty strings or 0 etc.

- '`isnull`' command returns the true value if any row of has null values. Since the rows 3-4 has NaN value, therefore, these are displayed as True.

```
c = casts
c['n'].isnull().head()
```

```
0 False
1 False
2 False
3 True
4 True
Name: n, dtype: bool
```

- ‘notnull’ is opposite of isnull, it returns true for not null values,

```
c['n'].notnull().head()

0 True
1 True
2 True
3 False
4 False
Name: n, dtype: bool
```

- To display the rows with null values, the condition must be passed in the DataFrame,

```
c[c['n'].isnull()].head(3)

     title             year     name type   character      n
3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NaN
4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron   NaN
5 Straight Outta Compton 2015 $hutter actor Club Patron   NaN
```

- NaN values can be filled by using fillna, ffill(forward fill), and bfill(backward fill) etc. In the code below, ‘NaN’ values are replaced by NA.

```
c_fill = c[c['n'].isnull()].fillna('NA')
c_fill.head(2)


     title           year       name    type  character   n
3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NA
4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron   NA
```

String operations

Various string operations can be performed using ‘.str.’ option. Let’s search for the movie “Maa” first,

```
t = titles
t[t['title'] == 'Maa']
title year
38880 Maa 1968
```

There is only one movie in the list. Now, we want to search all the movies which starts with ‘Maa’. The ‘.str.’ option is required for such queries as shown below,

```
t[t['title'].str.startswith("Maa ")].head(3)

           title      year
19 Maa Durga Shakti   1999
3046 Maa Aur Mamta    1970
```

```
7470 Maa Vaibhav Laxmi 1989
```

Count Values

Total number of occurrences can be counted using ‘value_counts()’ option. In following code, the total number of movies are displayed base on years.

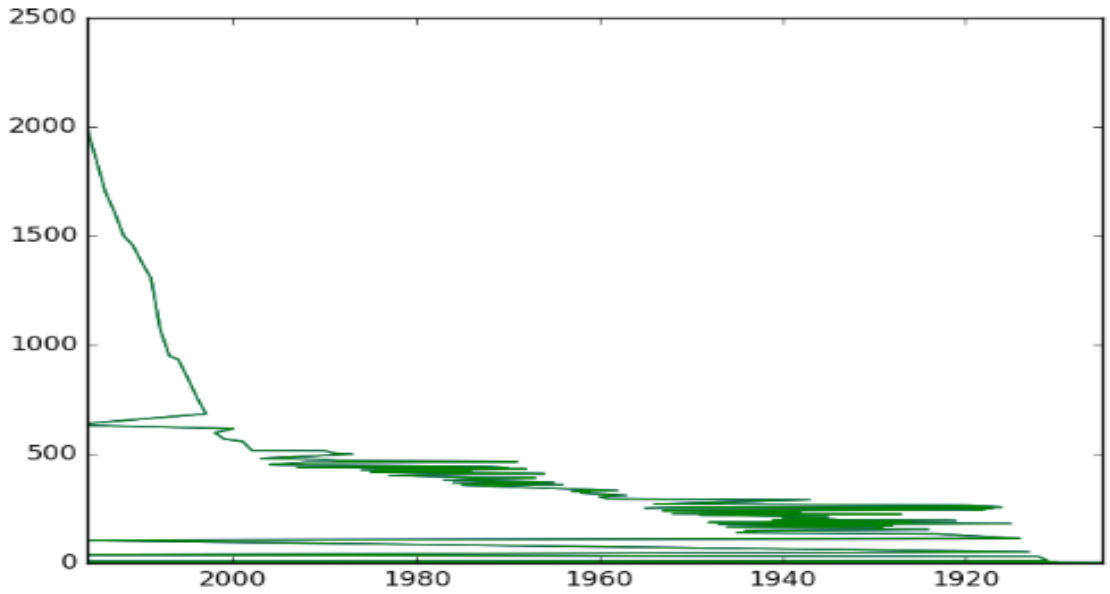
```
t['year'].value_counts().head()
```

```
2016 2363
2017 2138
2015 1849
2014 1701
2013 1609
Name: year, dtype: int64
```

Plots

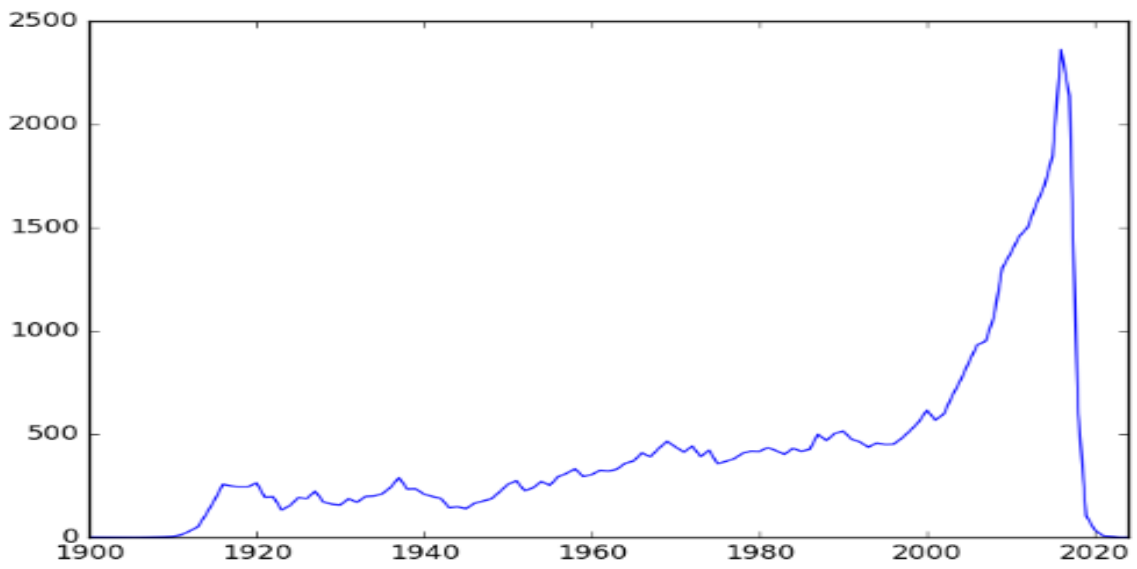
Pandas supports the matplotlib library and can be used to plot the data as well. In the previous section, the total numbers of movies/year were filtered out from the DataFrame. In the below code, those values are saved in new DataFrame and then plotted using panda,

```
import matplotlib.pyplot as plt
t = titles
p = t['year'].value_counts()
p.plot()
plt.show()
```



It’s better to sort the years (i.e. index) first and then plot the data as below. Here, the plot shows that number of movies is increasing every year.

```
p.sort_index().plot()
plt.show()
```



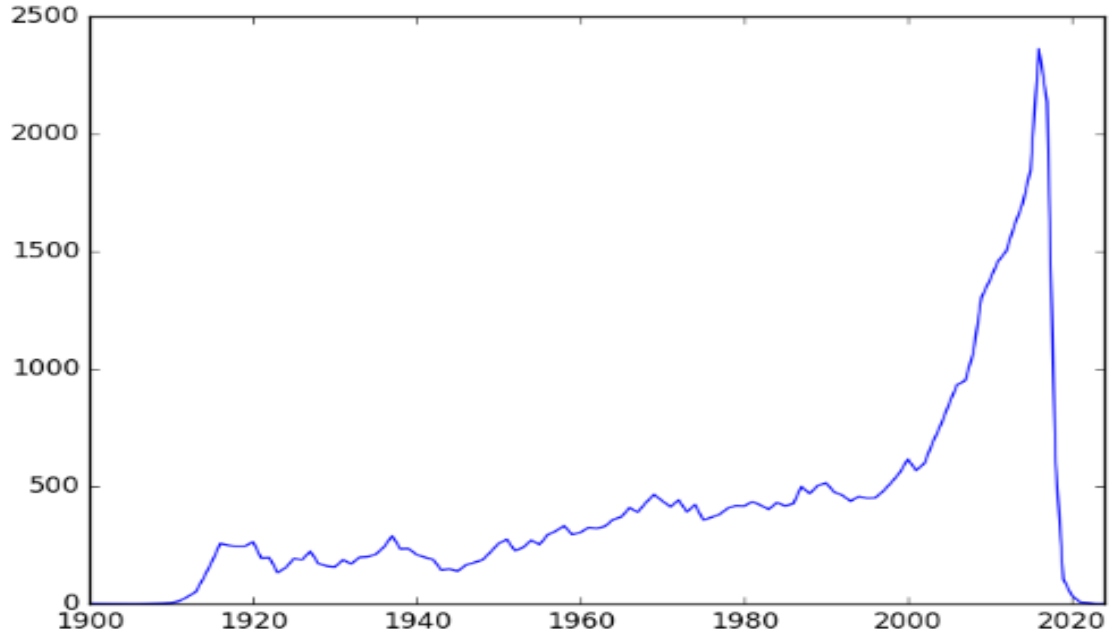
Groupby

Data can be grouped by columns-headers. Further, custom formats can be defined to group the various elements of the DataFrame.

Groupby with column-names

The value of movies/year was counted using 'count_values()' method. Same can be achieved by 'groupby' method as well. The 'groupby' command returns an object, and we need an additional functionality to it to get some results. For example, in the below code, data is grouped by 'year' and then size() command is used. The size() option counts the total number of rows for each year; therefore the result of the below code is the same as 'count_values()' command.

```
cg = c.groupby(['year']).size()
cg.plot()
plt.show()
```



- Further, groupby option can take multiple parameters for grouping. For example, we want to group the movies of the actor 'Aaron Abrams' based on year,

```
c = casts
cf = c[c['name'] == 'Aaron Abrams']
cf.groupby(['year']).size().head()
```

```
year
2003 2
2004 2
2005 2
2006 1
2007 2
dtype: int64
```

- Next, we want to see the list of movies as well, then we can pass two parameters in the list as shown below,

```
cf.groupby(['year', 'title']).size().head()
```

```
year        title
2003 The In-Laws 1
      The Visual Bible: The Gospel of John 1
2004 Resident Evil: Apocalypse 1
      Siblings 1
2005 Cinderella Man 1
```

```
dtype: int64
```

The groupby operation is performed on the ‘year’ first and then on ‘title’. In other words, first all the movies are grouped by year. After that, the result of this groupby is again grouped based on titles. Note that, the first group command arranged the year i.e. 2003, 2004 and 2005 etc.; then next group command arranged the title in alphabetical order.

- Next, we want to do grouping based on maximum ratings in a year; i.e. we want to group the items by year and see the maximum rating in those years,

```
c.groupby(['year']).n.max().head()
```

```
year
1912 6.0
1913 14.0
1914 39.0
1915 14.0
1916 35.0
Name: n, dtype: float64
```

- Similarly, we can check for the minimum rating,

```
c.groupby(['year']).n.min().head()
```

```
year
1912 6.0
1913 1.0
1914 1.0
1915 1.0
1916 1.0
Name: n, dtype: float64
```

- Lastly, we want to check the mean rating each year,

```
c.groupby(['year']).n.mean().head()
```

```
year
1912 6.000000
1913 4.142857
1914 7.085106
1915 4.236111
1916 5.037736
Name: n, dtype: float64
```

Merge

Usually, different data from the same project are available in various files. To get useful information from these files, we need to combine these files. Also, we need to merge to different data in the same file to get

some specific information. In this section, we will understand these two merges i.e. merge with different file and merge with the same file.

Merge with different files

we will merge the data of two tables i.e. 'release_dates.csv' and 'cast.csv'. The 'release_dates.csv' file contains the release date of movies in different countries.

- First, load the 'release_dates.csv' file, which contains the release dates of some of the movies, listed in 'cast.csv'. Following are the content of 'release_dates.csv' file,

```
release = pd.read_csv('release_dates.csv', index_col=None)
release.head()
```

```
        Title          year country      date
0 #73, ShaanthiNivaasa 2007 India        2007-06-15
1 #Beings               2015 Romania     2015-01-29
2 #Declimax             2018 Netherlands 2018-01-21
3 #Ewankosau saranghaeyo 2015 Philippines 2015-01-21
4 #Horror               2015 USA         2015-11-20
```

```
casts.head()
```

```
     title     year    name  type    character            n
0 Closet Monster 2015 Buffy #1 actor    Buffy 4          31.0
1 Suuri illusioni 1985 Homo $ actor Guests              22.0
2 Battle of the Sexes 2017 $hutter actor Bobby Riggs Fan 10.0
3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NaN
4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron NaN
```

- Let's see the release date of the movie 'Amelia'. For this first, filter out the Amelia from the DataFrame 'cast' as below. There are only two entries for the movie Amelia.

```
c_amelia = casts[casts['title'] == 'Amelia']
c_amelia.head()
```

```
      title year name type character            n
5767 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0
23319 Amelia 2009 Jeremy Akerman actor Sheriff 19.0
```

- Next, we will see the entries of movie 'Amelia' in release dates as below. In the below results, we can see that there are two different release years for the movie i.e. 1966 and 2009.

```
release [ release['title'] == 'Amelia' ].head()
```

```
title         year country date
20543 Amelia 1966 Mexico 1966-03-10
```

```
20544 Amelia 2009 Canada 2009-10-23
20545 Amelia 2009 USA 2009-10-23
20546 Amelia 2009 Australia 2009-11-12
20547 Amelia 2009 Singapore 2009-11-12
```

- Since there is no entry for Amelia-1966 in casts DataFrame, therefore merge command will not merge the Amelia-1966 release dates. In the following results, we can see that only Amelia 2009 release dates are merged with casts DataFrame.

```
c_amelia.merge(release).head()
```

```
         title year name type character n country date
0 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Canada 2009-10-23
1 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 USA 2009-10-23
2 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Australia 2009-11-12
3 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Singapore 2009-11-12
4 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Ireland 2009-11-13
```

Merge table with itself

Suppose, we want see the list of co-actors in the movies. For this, we need to merge the table with itself based on the title and year, as shown below. In the below code, co-star for actor 'Aaron Abrams' are displayed,

- First, filter out the results for 'Aaron Abrams',

```
c = casts[ casts['name']=='Aaron Abrams' ]
c.head(2)
```

```
title year name type character n
5765 #FromJennifer 2017 Aaron Abrams actor Ralph Sinclair NaN 5766
388 Arletta Avenue 2011 Aaron Abrams actor Alex 4.0
```

- Next, to find the co-stars, merge the DataFrame with itself based on 'title' and 'year' i.e. for being a co-star, the name of the movie and the year must be same,
- Note that 'casts' is used inside the bracket instead of c.

```
c.merge(casts, on=['title', 'year']).head()
```

The problem with the above joining is that it displays Aaron Abrams' as his co-actor as well (see first row). This problem can be avoided as below,

```
c_costar = c.merge (casts, on=['title', 'year'])
c_costar = c_costar[c_costar['name_y'] != 'Aaron Abrams']
c_costar.head()
```

Note: Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc. Increasingly, packages are being built on top of pandas to address specific needs in data preparation, analysis and visualization. This is encouraging

because it means pandas is not only helping users to handle their data tasks but also that it provides a better starting point for developers to build powerful and more focused data tools.

Python Packages
Module 2 : Numpy

Learning objectives

1. Define a data structure in Python (lists, numpy arrays).
2. Explain the structure (dimensionality) of numpy arrays.
3. Explain the differences between Python lists and numpy arrays.

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape

manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typi-cally, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing,and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most)of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example,consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

```
c = []for i in range(len(a)):c.append(a[i]*b[i])
```

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

```
for (i = 0; i < rows; i++):
{c[i] = a[i]*b[i];}
```

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

```
for (i = 0; i < rows; i++):
{for (j = 0; j < columns; j++):
{c[i][j] = a[i][j]*b[i][j];}}
```

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when anndarrayisinvolved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

```
c = a*b
```

does what the earlier examples do, at near-C speeds, but with the code simplicity, we expect from something based on python. Indeed, the NumPy idiom is even up simpler! This last example illustrates two of NumPy’s features which are the basis of much of its power: vectorization and broadcasting. Vectorization describes the absence of any explicit looping, indexing, etc.in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

- vectorized code is more concise and easier to read
- fewer lines of code generally mean fewer bugs
- the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
- vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with an efficient and difficult to read for loops.

NumPy fully supports an object-oriented approach, starting, once again, with ndarray. For example,ndarray is a class, possessing numerous methods and attributes. Many of its methods mirror functions in the outermost NumPynamespace, giving the programmer complete freedom to code in whichever paradigm she prefers and/or which seems most appropriate to the task at hand.

The Basics

NumPy’s main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy dimensions are called axes. The number of axes is rank.

For example, the coordinates of a point in 3D space[1, 2, 1]is an array of rank 1, because it has one axis. That axis has a length of 3. In the example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis)has a length of 2, the second dimension has a length of 3.

```
[[ 1., 0., 0.],
[ 0., 1., 2.]]
```

Numpy’s array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an ndarray object are:

`Ndarray.ndim` the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as rank.

`Ndarray.shape` the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be(n,m). The length of theshapetuple is therefore therank, or number of dimensions,ndim.

`Ndarray.size` the total number of elements of the array. This is equal to the product of the elements of shape.

`Ndarray.dtype` an object describing the type of the elements in the array. One can create or specify dtype using standardPython types. Additionally NumPy provides types of its own. `numpy.int32`, `numpy.int16`, and `numpy.float64` are some examples.

`Ndarray.itemsize` the size in bytes of each element of the array.For example, an array of elements of type `float64` has itemsize8 (=64/8), while one of type complex32 has itemsize4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

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

Example

```
import numpy as np
A = np.arange(15).reshape(3, 5)
print(a)

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

A.shape
(3,5)

A.ndim
2

A.dtype.name
'Int64'

A.itemsize
8

A.size
15

import numpy as np
a = np.array([2,3,4])
```

```
a
array([2, 3, 4])

a.dtype
dtype('int64')

b = np.array([1.2, 3.5, 5.1])
b.dtype
dtype('float64')
```

A frequent error consists in calling array with multiple numeric arguments, rather than providing a single list of numbers as an argument. Array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64.

```
s = np.zeros((3,4))
print(s)

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

s = np.ones((4,4))
print(s)

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

To create sequences of numbers, NumPy provides a function analogous torangethat returns arrays instead of lists

```
np.arange( 10, 30, 5 )
array([10, 15, 20, 25])

np.arange( 0, 2, 0.3 )
array([ 0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

Arithmetic operators on arrays apply element wise. A new array is created and filled with the result.

```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
b
array([0, 1, 2, 3])
c = a-b
c
array([20, 29, 38, 47])
b**2
array([0, 1, 4, 9])
10*np.sin(a)
array([ 9.12945251, -9.88031624, 7.4511316 , -2.62374854])
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

```
a = np.random.random((2,3))
a
array([[    0.18626021,    0.34556073,    0.39676747],[    0.53881673,
0.41919451, 0.6852195 ]])

a.sum()
2.5718191614547998

a.min()
0.1862602113776709

a.max()
0.6852195003967595
```

these operations apply to the array as though it were a list of numbers, regardless of its shape. However,by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
b = np.arange(12).reshape(3,4)
b
array([[ 0, 1, 2, 3],[ 4, 5, 6, 7],[ 8, 9, 10, 11]])

b.sum(axis=0)                 # sum of each column
array([12, 15, 18, 21])

b.min(axis=1)                 # min of each row
array([0, 4, 8])

b.cumsum(axis=1)              # cumulative sum along each row
array([[ 0, 1, 3, 6],[ 4, 9, 15, 22],[ 8, 17, 27, 38]])
```

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called "universal functions"(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
B = np.arange(3)
B
array([0, 1, 2])

np.exp(B)
array([ 1. , 2.71828183, 7.3890561 ])

np.sqrt(B)
array([ 0. , 1. , 1.41421356])

C = np.array([2., -1., 4.])
np.add(B, C)
array([ 2., 0., 6.])
```

Shape Manipulation

```
a = np.floor(10*np.random.random((3,4)))
a
array([[ 2., 8., 0., 6.],[ 4., 5., 1., 1.],[ 8., 9., 3., 6.]])

a.shape
(3, 4)
```

The shape of an array can be changed with various commands:

```
a.ravel()                  # flatten the array
array([ 2., 8., 0., 6., 4., 5., 1., 1., 8., 9., 3., 6.])
a.shape = (6, 2)
a.T
array([[ 2., 0., 4., 1., 8., 3.],[ 8., 6., 5., 1., 9., 6.]])
```

The order of the elements in the array resulting from `ravel()` is normally "C-style", that is, the rightmost index "changes the fastest", so the element after a[0,0] is a[0,1]. If the array is reshaped to some other shape, again the array is treated as "C-style". Numpy normally creates arrays stored in this order, so `ravel()` will usually not need to copy its argument,but if the array was made by taking slices of another array or created with unusual options, it may need to be copied.The functions `ravel()` and `reshape()` can also be instructed, using an optional argument, to use FORTRAN-style arrays,in which the leftmost index changes the fastest.

Their shape function returns its argument with a modified shape, whereas the ndarray.resize method modifies the array itself:

```
a
array([[ 2., 8.],[ 0., 6.],[ 4., 5.],[ 1., 1.],[ 8., 9.],[ 3., 6.]])

a.resize((2,6))
```

```
a
array([[ 2., 8., 0., 6., 4., 5.],[ 1., 1., 8., 9., 3., 6.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
a.reshape(3,-1)
array([[ 2., 8., 0., 6.],[ 4., 5., 1., 1.],[ 8., 9., 3., 6.]])
```

Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are three cases:

Simple assignments make no copy of array objects or of their data.

```
a = np.arange(12)
b = a                          # no new object is created
Bisa                               # a and b are two names for the same
ndarray objectTrue
b.shape = 3,4                  # changes the shape of a
a.shape(3, 4)
```

Different array objects can share the same data. The view method creates a new array object that looks at the same data.

```
c = a.view()
c is a
False
C.baseis a                     # C is a view of the data owned by aTrue
C.flags.owndata
False
c.shape = 2,6                  # a's shape doesn't change
a.shape(3, 4)
c[0,4] = 1234                  # a's data changes
aarray([[ 0, 1, 2, 3],[1234, 5, 6, 7],[ 8, 9, 10, 11]])
```

Functions and Methods

ArrayCreation

```
Arange,array,copy,empty,empty_like,eye,fromfile,fromfunction,identity
,linspace,logspace,mgrid,ogrid,ones,ones_like,r,zeros,zeros_like
```

Conversions

```
Ndarray.astype,atleast_1d,atleast_2d,atleast_3d,mat
```

Manipulations

Array_split,column_stack,concatenate,diagonal,dsplit,dstack,hsplit,st
ack,ndarray.item,newaxis,ravel,repeat,reshape,resize,squeeze,swapaxes
,take,transpose,vsplit,vstack

Questions
all,any,nonzero,where

Ordering
Argmax,argmin,argsort,max,min,ptp,searchsorted,sort

Operations
Choose,compress,cumprod,cumsum,inner,ndarray.fill,imag,prod,put,
putmask,real,sum

Basic Statistics
cov,mean,std,var

Basic Linear Algebra
Cross,dot,outer,linalg.svd,vdot

Linear Algebra

```
#Simple Array Operations

import numpy as np
a = np.array([[1.0, 2.0], [3.0, 4.0]])
print(a)
[[ 1. 2.][ 3. 4.]]

a.transpose()
array([[ 1., 3.],[ 2., 4.]])

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

u = np.eye(2)           # unit 2x2 matrix; "eye" represents "I"
u
array([[ 1., 0.],[ 0., 1.]])

j = np.array([[0.0, -1.0], [1.0, 0.0]])
np.dot (j, j)           # matrix product
array([[-1., 0.],[ 0., -1.]])

np.trace(u)             # trace2.0

y = np.array([[5.], [7.]])
```

```
np.linalg.solve(a, y)
array([[-3.],[ 4.]])
np.linalg.eig(j)

(array([ 0.+1.j,  0.-1.j]), array([[ 0.70710678+0.j  ,  0.70710678-0.j
],[ 0.00000000-0.70710678j, 0.00000000+0.70710678j]]))
```

Parameters:square matrixReturnsThe eigenvalues, each repeated according to its multiplicity.The normalized (unit "length") eigenvectors, such that the column ``v[:,i]`` is the eigenvector corresponding to the eigenvalue ``w[i]

Data types

Numpy supports a much greater variety of numerical types than Python does. This section shows which are available,and how to modify an array's data-type.

Additionally to intc the platform dependent C integer types short,long,longlong and their unsigned versions are defined. Numpy numerical types are instances of dtype(data-type) objects, each having unique characteristics. Once you have imported NumPy using

```
import numpy as np
```

the dtypes are available asnp.bool_,np.float32, etc.

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C `long`; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for `complex128`.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

There are 5 basic numerical types representing booleans (bool), integer (int), unsigned integers (uint) floating point(float) and complex. Those with numbers in their names indicate the bitsize of the type (i.e. how many bits are needed to represent a single value in memory). Some types, such asintandintp, have differing bit sizes, dependent on the platforms (e.g. 32-bit vs. 64-bit machines). This should be taken into account when interfacing with low-level code(such as C or Fortran) where the raw memory is addressed.
Data-types can be used as functions to convert python numbers to array scalars (see the array scalar section for an explanation), python sequences of numbers to arrays of that type, or as arguments to the dtype keyword that many numpy functions or methods accept. Some examples:

```
import numpy as np
x = np.float32(1.0)
x1.0
y = np.int_([1,2,4])
y
array([1, 2, 4])
z = np.arange(3, dtype=np.uint8)
z
array([0, 1, 2], dtype=uint8)
```

Array types can also be referred to by character codes, mostly to retain backward compatibility with older packages such as Numeric. Some documentation may still refer to these, for example:

```
np.array([1, 2, 3], dtype='f')
array([ 1., 2., 3.], dtype=float32)
```

Array Scalars

Numpy generally returns elements of arrays as array scalars (a scalar with an associated dtype). Array scalars differ from Python scalars, but for the most part they can be used interchangeably (the primary exception is for versions of Python older than v2.x, where integer array scalars cannot act as indices for lists and tuples). There are some exceptions, such as when code requires very specific attributes of a scalar or when it checks specifically whether a value is a Python scalar. Generally, problems are easily fixed by explicitly converting array scalars to Python scalars,using the corresponding Python type function (e.g.,int,float,complex,str,unicode).The primary advantage of using array scalars is that they preserve the array type (Python may not have a matching scalar type available, e.g. int16). Therefore, the use of array scalars ensures identical behaviour between arrays and scalars, irrespective of whether the value is inside an array or not. NumPy scalars also have many of the same methods arrays do.

Note: Pythons NumPy library is one of the most popular libraries for numerical computing. We explored the NumPy library in detail with the help of several examples. The amount of options it provides and the wide array of things one can do with simple statements is magnificent and really amazing.