Pandas :

we have so far completed modules :

1) Random

2) numpy

* Pandas is built on top of numpy.
* Pandas are used for data analysis and machine learning more than numpy.
* Pandas consist of three main data structures.
  - Series
  - Data Frame
  - Panel

with help of these we can perform high performance multi dimensional manipulation.

* Data Analysis mainly uses pandas because of three data structures.
* Panel is not used much.
* main focus is series & Data Frame.

Pandas installation

```
pip install pandas
```

import pandas module

```
import pandas as pd
```

import numpy module (optional)

```
import numpy as np
```

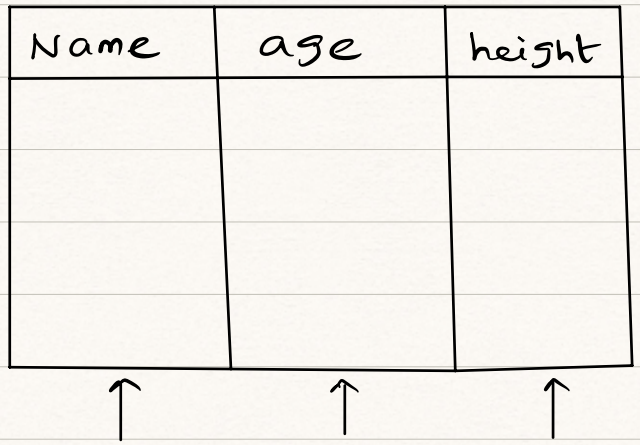
Series - first data structure we will learn.

datastructure - structure of data.

Properties of Series :-

1) It is a pandas data structure.
2) Series is one dimensional labelled array.
3) Series can only hold homogenous data.
4) Data inside the series are mutable.
5) Series will hold duplicate values.
6) Series are ordered.

* Imagine Series as single column for easy understanding.



3 series with same type of data

How to create series

* Pandas we can apply our own index label to the arrays. (numpy cannot do it)
  default is : 0, 1, 2, ... len(object)
* Own index label can be anything.

To create series we use Series() method with 'S' capital.

* It will take few parameters, (data or object), index (our own index or default index), dtype can be python or panda (numpy) dtype.
* data is usually passed as single list.

```
pd.Series(object(or data), index, dtype)
```

* Series function is returning a series object

To create an Empty series

```
pd.Series()
  ↳ Series([], dtype=float64)

x= pd.Series()
  ↳ Series Object
type(x)
  ↳ pandas.core.series.Series
```

Create series with values in it.

result will show index and actual value.

```
pd.Series([1,2,3,4])
↳
0   1
1   2
2   3
3   4
dtype: int64
```

Index → 0 1 ⟶ Value

If a float element is present in data then float takes preference.

If we want to preserve data type we can convert to object data type

```
pd.Series([1,2,3.2,4])
↳ 0  1.0
  1  2.0
  2  3.2
  4  4.0
  dtype: float64
```

```
y = pd.Series([1,2,3.2,4])
y
↳ 0  1.0
  1  2.0
  2  3.2
  4  4.0
  dtype: float64
```

```
z = pd.Series([1, 2, 3.2, 4], dtype=complex)
```

```
z
↳ 0  1.0 +0.0j
  1  2.0 +0.0j
  2  3.2 +0.0j
  4  4.0 +0.0j
  dtype: complex128
```

if we change dtype to 'object' data integrity is maintained.

```
s = pd.Series([1, 2, 3.2, 4], dtype=object)
```

we can give our own index values.

no.of indexes should always match the length of the values.

```
pd.Series([1, 2, 3.2, 4], index=["a", "b", "c", "d"],
                                  dtype=object)
```

```
↳ a  1
  b  2
  c  3.2
  d  4
  dtype: object
```

we can create series using numpy array

```
pd.Series(np.array([1,2,3,4]))
```

↳

```
0  1
1  2
2  3
3  4
dtype: int32
```

we can use numpy functions to create series.

```
pd.Series(np.zeros((4,)))
```

↳

```
0  0.0
1  0.0
2  0.0
3  0.0
dtype: float64
```

Series is 1-dimensional.

data given cannot be more than 1-dimensional.

```
pd.Series(np.array([[1,2,3,4]]))
```

↳ X error (data cannot be 2-D)

we can use tuples to create series.

```
pd.Series((1,2,3,4), index=('a','b','c','d'), dtype=object)
```

↳

```
a  1
b  2
c  3
d  4
dtype= object
```

set type is unordered so we cannot use set to create series data structure object.

```
pd.Series({1,2,3,4}, index=('a','b','c','d'), dtype=object)
```

↳ X (error as sets are unordered and series is labelled data structure)

using dictionary we can create Series. Labels will be key and data will be values.

```
pd.Series({'A':1,'B':2,'C':3})
```

↳

```
A 1
B 2
C 3
dtype= int64
```

if we don't mention index default it is using is

```
pd.Series([1,2,3,4], index=np.arange(len([1,2,3,4])))
```

↳

```
0   1
1   2
2   3
3   4
dtype: int64
```

if we are using our own index

```
pd.Series([1,2,3,4], index=['A','B','C','D'])
```

↳

```
A   1
B   2
C   3
D   4
dtype: int64
```

when using indexes not in Series created using dictionaries.

```
pd.Series({'A':1, 'B':2, 'C':3}, index=['a','b','c','A','B'])
```

↓

```
a   NaN
b   NaN
c   NaN
A   1.0
B   2.0
dtype: float64
```

* In this case for indexes not in Series data new elements are created and NaN (Null) value is assigned to it.
* All the elements are converted to float as NaN is float.

* Series will be created for indexes mentioned.
* So if we want to skip any elements during series creation from data we can skip their indexes.

```
pd.Series({'A':1, 'B':2, 'C':3}, index=['a','b','c','A','B'])
```

↓

```
a    NaN
b    NaN
c    NaN
A    1.0
B    2.0
dtype: float64
```

```
pd.Series({'A':1, 'B':2, 'C':3}, index=['a','b','c','A','B','C'])
```

↳

```
a    NaN
b    NaN
c    NaN
A    1.0
B    2.0
C    3.0
dtype: float64
```

Indexing and slicing can be used:

```
x = pd.Series([1, 2, 3, 2])
```

```
x
```

↳

```
0  1
1  2
2  3
3  2
dtype: int64
```

```
x[0]
```

↳ 1

```
x[-1]
  ↘ error -1 not in range
```

* whenever we are using integer based indexing negative indexing will not be allowed.
* In slicing negative indexing can be used

```
x[1]
  ↘ 2

x[0:2]
  ↘  0  1
     1  2
     dtype: int64

x[:-1]
  ↘  0  1
     1  2
     2  3
     dtype: int64
```

* default index is always 0, 1, 2, 3, ... len(data), the label we apply on it is only temporary.

```
y = pd.Series([1,2,3,12], index=['a','b','c','d'])

y[0]
  ↘ 1

y['a']
  ↘ 1
```

* Index values when duplicate are used we will not get error but it is not recommended as

when we are accessing it or manipulating it we will apply on multiple elements.

integer based indexing is used.

```
y[[0,1,2]]
  a  1
  b  2
  c  3
dtype: int64
```

```
y[['a','b','c']]
  a  1
  b  2
  c  3
dtype: int64
```

* When we are using strings as our personal index they work differently.
* All the values of the starting index and ending index including the ending index is returned.

```
y['a':'c']
  a  1
  b  2
  c  3
dtype= int64
```

* ending value is not included when default indexing is used.

```
y[0:3]
↳ a  1
  b  2
  c  3
  dtype = int64
```

* whenever we are using integers as personal index values then during slicing default indexes are considered not the personal indexes.
* when integers are used as personal index then ending value is not included.

```
y = pd.Series([1,2,3,12], index=[1,2,3,4])

y
↳ 1   1
  2   2
  3   3
  4   12
```

(default index
  0
  1
  2
  3)

```
y[1:3]
↳ 2   2
  3   3
  dtype = int64
```

(default index values are used here)

(ending value i.e default y[3] not included)

```
y = pd.Series([1,2,3,12], index=[11,12,13,14])
```

```
y
 ↳ 11   1
   12   2
   13   3
   14   12
```

(default index)

0
1
2
3

```
y[11]
 ↳ 1
y[11:13]
 ↳ Series[]  (empty series)
```

(Since in default indexes we only have until 3 so 11 & 13 do not exist)

* For series it is better to use string index.
* Real world scenario rows are high in data compared to columns.
  (ex: Excel we use alphabets as labels and rows as numbers)
* So logically for small data we can use our own index but huge data we will just go with default index.

we can replace elements from a series.

y[11] = "a" (adding with replacement)

* string is internally converted to object datatype when string is added to maintain data integrity.

y

11	a
12	b
13	c
14	12

dtype = object

adding element without replacement

y[15] = 'd'

* will check if element is existing. If it does not exist it will be added.

y

11	a
12	2
13	3
14	12
15	d

dtype = object

Delete from Series

del (keyword) - we can use delete keyword to delete from the series.

* only one element at a time can be deleted.

ex:-

```
S = pd.Series(['a','b','c','d'])

S
↳ 0  a
  1  b
  2  c
  3  d
  dtype = object

S[2]
↳ 'c'

del S[2]

S
↳ 0  a
  1  b
  3  d
  dtype = object

del S[0:2]
↳ X error as only one element at a time can be deleted.
```

drop(): drop() is a series and DataFrame function mainly used to delete an element.

* Input will be a list of indexes
* It is a temporary function by default i.e the original series is not changed.

```
S
↳
0   a
1   b
2   c
3   d
dtype = object
```

```
S.drop([0,1])
↳
2   c
3   d
dtype = Object
```

this is temporary if we want to use result we have to save it to a variable.

```
S
↳
0   a
1   b
2   c
3   d
dtype = object
```

* If we want to do operation on the original series we can use a parameter called inplace.

* inplace decides if the manipulation should be directly performed on the original data set or make a copy and perform on it.

default : inplace = False

(i.e manipulation is performed on a copy of the series)

inplace = True → operation performed directly on the series.

```
q = s.drop([3], inplace = False)
```

q

```
0  a
1  b
2  c
dtype: object
```

(changed)

s

```
0  a
1  b
2  c
3  d
dtype: object
```

(unchanged)

```
s.drop([3], inplace = True)
```

s

```
0  a
1  b
2  c
dtype: object
```

(original array changed)

* operation on the series is done element wise.

Arithmetic operations

ex:- `pd.Series([1, 2, 3]) + 2`

```
0  3
1  4
2  5
dtype: int64
```

`pd.Series([1, 2, 3]) * 2`

```
0  2
1  4
2  6
dtype: int64
```

```
a = pd.Series([1, 2, 3])
b = pd.Series([4, 5, 6])
```

`a+b+a`

```
0  6
1  9
2  12
dtype = int64
```

```
C = pd.Series([4, 5])
```

`a+b+C`

```
0  9.0
1  12.0
2  NaN
dtype = float
```

(when there is a missing element NaN is placed in its place)

Comparision Operators (returns boolean series)

```
a > b
↳ 0   False
  1   False
  2   False
  dtype= bool
```

```
a == b
↳ 0   False
  1   False
  2   False
  dtype : bool
```

Problem

If `a = pd.Series([1, 2, 3])`
`b = pd.Series([4, 5, 6])`

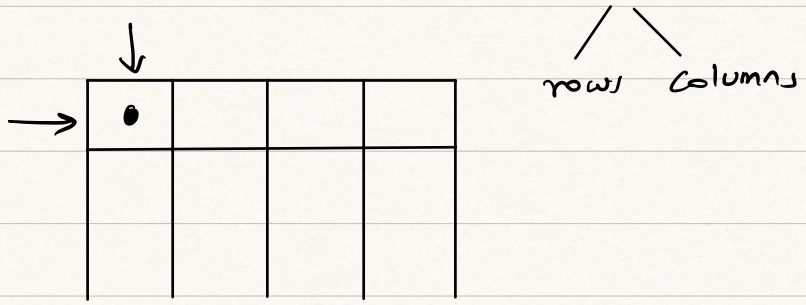
$a^b = ?$

```
a ** b
↳ 0   1
  1   32
  3   729
  dtype : int64
```

DataFrame

A pandas DataFrame is a 2-Dimensional datastructure like a 2 dimensional array, or a table with rows and columns.

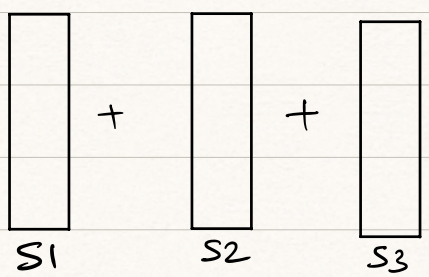
DataFrame is a 2-Dimensional labelled array.



* we can give our own labels to rows (Indexes) and columns (Columns).

Otherways to look at DataFrame

└ It is a combination of series



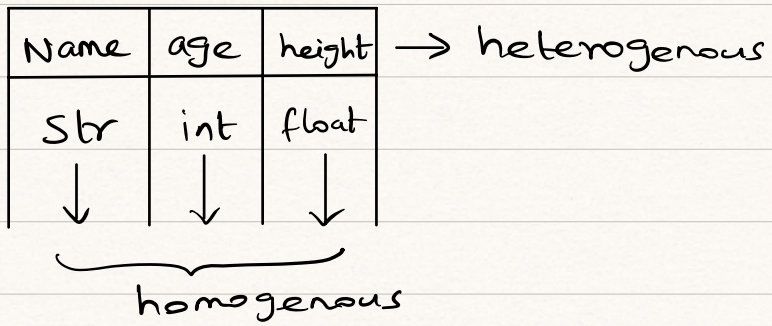
↘ DataFrame

DataFrame

- Container which is holding multiple series
- extension of series.

Properties of DataFrame

1) DataFrame is a panda datastructure.
2) DataFrame is a 2-Dimensional labelled array.
3) Rows in the DataFrame are heterogenous whereas the columns are homogenous.



4) DataFrames are mutable.
   Values inside are also mutable.
5) we can change the shape of the DataFrame.
   i-e mxn or (m,n) can be changed to (n,m)
   * Series shape cannot be changed.

Create a DataFrame

pd.DataFrame(object(or data), index=[...], columns=[...], dtype).

in DataFrame both D and F are capital.

index = [...] are labels for rows
(generally not used)

columns = [...] are custom labels for columns
(used more frequently)

* whenever we pass a single list as data we will get a single column which is like a series.
* dtype will be applied column wise.

```
pd.DataFrame([1, 2, 3, 4])
```

	0
0	1
1	2
2	3
3	4

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

0 1
1 2
2 3
3 4
dtype: int64

difference in appearance

if we want to apply custom column labels & row labels.

```
pd.DataFrame([1, 2, 3, 4], index=['a', 'b', 'c', 'd'],
                                    columns=['A'])
```

	A
a	1
b	2
c	3
d	4

* Index should match column length otherwise will get an error.

```
pd.DataFrame([1, 2, 3, 4], index=['a', 'b', 'c', 'd'],
                                    columns=['A'])
```

→ X shape of passed values (4, 1)
shape of indexes (3, 1)

```
pd.DataFrame([[1, 2, 3, 4]])
```

	0	1	2	3
0	1	2	3	4

* when we are using double square bracket notation. The inner square brackets will become rows.

```
pd.DataFrame([[1, 2, 3, 4], [5, 6, 7, 8], [7, 1, 2, 3]])
```

↳

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	7	1	2	3

* For missing elements NaN is used in DataFrame and DataFrame is converted to float.

pd.DataFrame([[1,2,3,4],[5,6,7],[7,1,2,3]])

↳

	0	1	2	3
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	NaN
2	7.0	1.0	2.0	3.0

NaN - not a number
dtype = float.

pd.DataFrame([[1,2,3,4,'5'],[5,6,7,'T'],[7,1,2,3,'D']], columns = ['A','B','C','D','E'])

↳

	A	B	C	D	E
0	1	2	3	4	5
1	5	6	7	T	None
2	7	1	2	3	D

None - Python Singelton object
dtype = object

when we try to explicitly convert string to float it will just skip that data and not give any error.

```
pd.DataFrame([[1,2,3,4,'S'],[5,6,7,'T'],[7,1,2,3,'D']],
             columns=['A','B','C','D','E'],dtype=float)
```

↳

	A	B	C	D	E
0	1.0	2.0	3.0	4.0	S
1	5.0	6.0	7.0	T	None
2	7.0	1.0	2.0	3.0	D

* As data to DataFrame( ) we can use list of list, tuple of list and list of tuple.
* we cannot use sets as sets are unhashable.
* we can use arrays to create DataFrames.
* we can use dictionaries to create DataFrames.

```
pd.DataFrame(([1,2,3,'S'],[5,6,7,'T'],[7,1,2,'D']),
             columns=['A','B','C','D'],dtype=float)
```

↳

	A	B	C	D
0	1.0	2.0	3.0	S
1	5.0	6.0	7.0	T
2	7.0	1.0	2.0	D

```
import numpy as np

pd.DataFrame(([1,2,3,'S'],[5,6,7,'T'],[7,1,2,'D']),
             columns=['A','B','C','D'])
```

	0	1	2	3
0	1	2	3	5
1	5	6	7	T
2	7	1	2	D

using dictionaries

Keys - Columns label

Values - Column values

```
pd.DataFrame({'Name': ['P1', 'P2', 'P3']})
```

	Name
0	P1
1	P2
2	P3

Name ← key

} values

```
pd.DataFrame({'Name': ['P1', 'P2', 'P3'], 'Age': [23, 43, 12],
              'Height': [154.0, 170.0, 160.0]}, index=['a', 'b', 'c'])
```

	Name	Age	Height
0	P1	23	154.0
1	P2	43	170.0
2	P3	12	160.0

* we can mention the columns parameter also and order will be based on order of the columns mentioned.

* Only columns mentioned will be present in the DataFrame.
* If column labels not present in the data are mentioned. New column will be created with NaN values.
* we can add new values to these columns.

```
pd.DataFrame({'Name': ['P1', 'P2', 'P3'], 'Age': [23, 43, 12],
              'Height': [154.0, 170.0, 160.0]},
              columns = ['Name', 'Height', 'weight'])
```

↳

	Name	Height	weight
0	P1	154.0	NaN
1	P2	170.0	NaN
2	P3	160.0	NaN

→ order as mentioned in columns (Age excluded)

* DataFrame creates DataFrame Object.

i.e

```
df= pd.DataFrame({'Name': ['P1', 'P2', 'P3'], 'Age': [23, 43, 12],
                  'Height': [154.0, 170.0, 160.0]},
                  columns = ['Name', 'Height', 'weight'])
```

```
type(df)
```

↳ pandas.core.frame.DataFrame

* Accessing elements column wise where whole column is returned.

This column will behave as Series.

* To access rows further functionality is needed that is taught later.

```
df["Age"]
```

```
0    23
1    43
2    12
Name: Age, dtype: int64
```

```
type(df['Age'])
```

```
pandas.core.series.Series
```

* we can access multiple elements

```
df[['Age', 'Name']]
```

	Age	Name
0	23	P1
1	43	P2
2	12	P3

* Add with replacement

It will check if already existing. If it is then values are modified.

```
df["Age"] = [0,0,0]
```

← Single bracket

← new values are assigned

df

	Name	Height	weight	Age
0	P1	154.2	NaN	0
1	P2	170.0	NaN	0
2	P3	169.0	NaN	0

* Add without replacement

```
df["ph_no"] = [123, 456, 789]
```

df

	Name	Height	weight	Age	ph_no
0	P1	154.2	NaN	0	123
1	P2	170.0	NaN	0	456
2	P3	169.0	NaN	0	789

* when adding 2 values we need to give double brackets.

```
df[['city', 'add']] = [[1, 2], [3, 4], [5, 6]]
```

df

	Name	Height	weight	Age	ph_no	city	add
0	P1	154.2	NaN	0	123	1	2
1	P2	170.0	NaN	0	456	3	4
2	P3	169.0	NaN	0	789	5	6

* How to drop columns
* drop( ) function in DataFrames needs axis parameter drop(axis)
* default is axis=0 row wise
  axis = 0 row wise (different from numpy)
  axis= 1 column wise
* we should pass single list not double list.

```
df.drop(['Age','Height'], axis=1)
```

↓

	Name	weight	ph_no	city	add
0	P1	NaN	123	1	2
1	P2	NaN	456	3	4
2	P3	NaN	789	5	6

```
df.drop(['Age', 'Height'])
```

↓ X (default is axis=0 rowwise 'Age' & 'Height' not present in rows)

```
df.drop([0])
```

↑ single bracket

↓

	Name	weight	ph_no	city	add
1	P2	NaN	456	3	4
2	P3	NaN	789	5	6

Advanced functionalities

```
df["height+10"] = df[Height]+10
```

	Name	Height	weight	Age	height+10
0	P1	154.2	23	23	164.2
1	P2	170.0	34	43	180.0
2	P3	169.0	45	12	179.0

Assignment

- Create a DataFrame with 5 columns
  - Name
  - Age
  - English Marks
  - Hindi Marks
  - Science Marks

Calculate the percentage

```
marks = pd.DataFrame({'Name': [P1, P2, P3], 'Age': [13, 13, 12],
                      'English': [75, 60, 85], 'Hindi': [80, 70, 90],
                      'science': [85, 75, 95]})
```

marks

	Name	Age	English	Hindi	science
0	P1	13	75	80	85
1	P2	13	60	70	75
2	P3	12	85	90	95

marks["percentage"] = (marks['English'] + marks['Hindi'] + marks['science'])/3

marks

	Name	Age	English	Hindi	Science	Percentage
0	P1	13	75	80	85	80.00...
1	P2	13	60	70	75	68.33...
2	P3	12	85	90	95	90.00...

* append() method

* will check column names and if present will be appended.

```
x = pd.DataFrame([[1,2,3],[4,5,6]], columns=['a','b','c'])
y = pd.DataFrame([[11,12,13],[14,15,16]], columns=['a','b','c'])
```

x.append(y)

	a	b	c
0	1	2	3
1	4	5	6
0	11	12	13
1	14	15	16

Index Values

0
1 → copied
0
1

* if we want to ignore the original index when appending we use ignore_index parameter.

x.append(y, ignore_index = True)

	a	b	c
0	1	2	3
1	4	5	6
2	11	12	13
3	14	15	16

Index Values are new

* If DataFrames of different columns are added the missing values will have NaN values

z = pd.DataFrame([[11,12,13],[14,15,16]], columns = ['a','b','c'])

columns c & d are not common between x & z

x.append(y)

	a	b	c	d
0	1	2	3.0	NaN
1	4	5	6.0	NaN
0	11	12	NaN	13.0
1	14	15	NaN	16.0

Index Values: 0, 1 → copied, 0, 1

x.append(y, ignore_index = True)

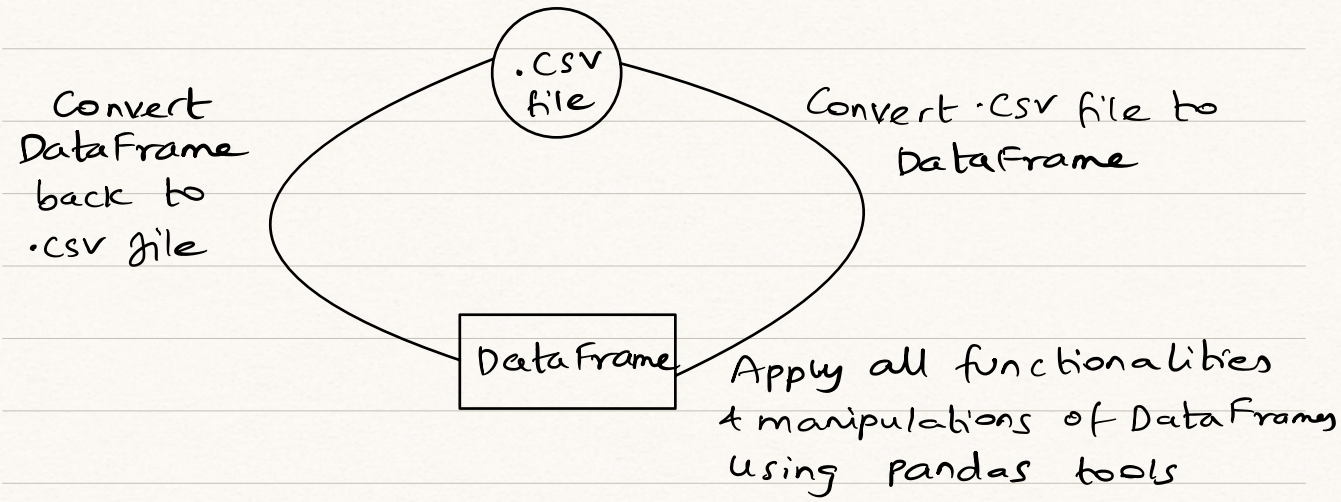
	a	b	c	d
0	1	2	3.0	NaN
1	4	5	6.0	NaN
2	11	12	NaN	13.0
3	14	15	NaN	16.0

Index Values are new

Importing .csv & .txt files using pandas

In real world scenario it is a huge task to create big DataFrames.

Instead we use .csv (excel single sheet) or .txt files and do manipulation.



- .csv file in Excel sheet looks exactly like DataFrame with rows and columns.

csv - comma seperated values

↓

seperator

ex:- 1, 2, 3, 4, 5, ...

we can have data file with other seperators like ':' colon and such.

* Before converting the given file we need to know the seperator.

* Function used in panda to read a .csv or .txt file is

pd.read_csv()

↳ DataFrame object

(reads a .csv or .txt file and returns DataFrame object

pd.read_csv("file_path", ...)

↓

there are many parameters for additional functionality that we will slowly learn.

Sep (seperator): Default is comma

If the data has a different seperator we can explicitly give it to read_csv() function.

ex:-

pd.read_csv(r"C:\....\Book1.csv", sep=",")

↳

	Name	Age	Gender	Height
0	P1	23	m	153.2
1	P2	21	f	123.1
2	P3	58	f	173.8
3	P4	24	m	123.2
4	P5	65	m	163.8
5	P6	12	f	121.7

* Excel files have .xlsx extension which is not exactly .csv files (as Excel is a collection of workbooks)
* single sheet can be used as .csv file. combination of .csv files can be called .xlsx file.

So **pd.read_csv()**
- cannot read .xlsx files
- can only read .csv or .txt file.

* So data in Excel needs to be saved as .csv or .txt files.

If data is ':' colon seperated we can use

```
pd.read_csv(r"C:\....\Book1.txt", sep = ":")
```

header :

In real world scenario data need not be clean so there might be scrap data at the start of the file.

* when .csv file is converted to DataFrame by default row '0' is considered as header (i.e column labels)

If first row or first few rows have scrap data we can explicitly mention which row can be

used as header by mentioning its index.

Rows above will be scrapped during DataFrame creation.

```
pd.read_csv("file_path", sep=";", header=0)
```

↑ default

If we want to consider row '1' as header then

```
pd.read_csv("file_path", sep=";", header=1)
```

* Empty cells in Excel when converted to DataFrame will have 'NaN' value.

Functions to manipulate DataFrames

head() - will return topmost number of rows mentioned.

default is 5 rows

DataFrame Object . head()

i.e

data = pd.read_csv("file-path", sep=';')

data.head()

↳ first 5 rows of csv file or DataFrame

data.head(2)

↳ first 2 rows of DataFrame

* If we want to this sub DataFrame we can save it and use it.

tail() - will return the bottom most rows mentioned.

default - 5 bottom rows

DataFrame Object . tail()

↳ bottom five rows of DataFrame

i.e data.tail(5)

↳ bottom 5 rows

data.tail(2)

↳ bottom 2 rows.

info() - Applying this to the DataFrame gives the entire information of the DataFrame.

DataFrame object.info()

i.e data.info()

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

RangeIndex: 6 entries, 0 to 5

Data columns (total 4 columns):

#	Column	Non-Null count	Dtype
0	Name	6 Non-Null	Object
1	Age	6 Non-Null	int64
2	Gender	6 Non-Null	Object
3	Height	6 Non-Null	float64

dtypes: float64(1), int64(1), object(2)

memory usage: 320.0+ bytes

* with the help of this info we can learn many things about Data.

So here below are the points we deduce.

1) belongs to panda DataFrame.
2) Each row will be one entry containing all the information, index from 0-5.
3) Columns are 4
4) default column names, our column label, are there any non-Null values (i.e data values),

dtype of each column (string is considered as object)

5) number of different dtype of columns.

6) How much memory was used.

drop() : drops a column mentioned.

* It is highly recommended to create a copy of DataFrame and do manipulations on it.
* Better not use inplace as original DataFrame will be modified.
* we can also use data_new = data.copy() and use it.

```
new_data = data.drop(['Height'], axis = 1)
                                    ↑ column wise
```

If we do rowwise i.e axis=0 we will get error.

new_data

	Name	Age	Gender
0	P1	---	---
1	P2	---	---
2	P3	---	---
3	P4	---	---
4	P5	---	---
5	P6	---	---

Writing back to a .csv or .txt file

.to_csv (file-path)

we can save the manipulated DataFrame to a new .csv or .txt file.

```
new_data.to_csv(r"C:\....\Book3.csv")
pd.read_csv(r"C:\....\Book3.csv")
```

↳

	Unnamed:0	Name	Age	Gender
0	0	P1	--	--
1	1	P2	--	--
2	2	P3	--	--
3	3	P4	--	--
4	4	P5	--	--
5	5	P6	--	--

* If we don't want the DataFrame index values to be copied to .csv file we can mention a parameter 'index=False'

i.e `new_data.to_csv(r"C:\....\Book4.csv", index=False)`

* Already existing file cannot be edited.

```
pd.read_csv(r"C:\....\Book4.csv")
```

↳

	Name	Age	Gender
0	P1	--	--
1	P2	--	--
2	.		
3	.		
4	.	--	--
5	.	--	--

Row accessing we have 2 attributes

- iloc (integer based location)
- loc (labelled based location)

loc is very important attribute.

iloc is not used much.

In Jupyter Notebook iloc & loc sometimes give some errors. Visual studio has no issues.

```
data1 = pd.DataFrame({'Name': ['P1', 'P2', 'P3', 'P4'], 'Age':
                      [12, 23, 42, 76]}, index=['a', 'b', 'c', 'd'])
```

data1 →

	Name	Age	default index
a	P1	12	0
b	P2	23	1
c	P3	42	2
d	P4	76	3

* In iloc we cannot use labels it only uses integer default indexes

```
data1.iloc["a":"c"]
```

→ X error

* we need to use integer default indexes

* As default indexing in slicing ending value is not included.

```
data1.iloc[0:2]
```

	Name	Age
a	P1	12
b	P2	23

```
data1.iloc[3]
```

```
Name 74
Age 76
Name d, dtype= object
```

* loc can use our own label.
* Since using labels ending value is included.

```
data2=pd.DataFrame({'Name':['P1','P2','P3','P4'],'Age':
                    [12,23,42,76]}, index=[1,2,3,4])
```

data2

	Name	Age
1	P1	12
2	P2	23
3	P3	42
4	P4	76

default index
0
1
2
3

* For slicing labels can be used.

```
data2.loc[1:3]
```

↳

	Name	Age
1	P1	12
2	P2	23
3	P3	42

* For indexing it will check if label exists

```
data2.loc[6]
```

↳ X error label does not exist

```
data2.loc[4]
```

↳

```
Name P4
Age 76
Name 4, dtype= object
```

Slicing

* with iloc we can do slicing, it will use default indexes for row and column

```
data2.iloc[0,1]
```

↳ 12

```
data2.iloc[1:,:]
```

↳

	Name	Age
2	P2	23
3	P2	42
4	P4	76

* Integer based indexing is generally not used in DataFrames.
* we will mainly use slicing based indexing.

```
data1.iloc[[1,1],[0,1]]
```

	Name	Age
b	P2	23
b	P2	23

integer based gives double like this

* Using loc we can do lots of manipulations.

data

	Name	Age	Gender	Height
0	P1	23	m	153.2
1	P2	21	f	123.1
2	P3	58	f	173.8
3	P4	24	m	123.2
4	P5	65	m	163.8
5	P6	12	f	121.1

Advanced functionalities

* If we want to get those rows where Height>143.
  - here we use loc as it is labelled as Height instead of 0,1,2,3,.... default values
  - here we access rows based on the condition.

* iloc cannot be used when we are accessing based on condition.

how to access rows based on a single column condition

loc is attribute

```
data.loc[data["Height"]>143]
```

↓

	Name	Age	Gender	Height
0	P1	23	M	153.2
2	P3	58	F	173.8
4	P5	65	M	163.8

when we want to access rows based on multiple conditions.

we use paranthesis between conditions with & (logical_and) and | (logical_or) in between them.

we want People of 'Height' > 143 and male candidates

```
data.loc[(data["Height"]>143)&(data["Gender"]=='m')]
```

↘

	Name	Age	Gender	Height
0	P1	23	m	153.2
4	P5	65	m	163.8

* This works without loc also but don't use as label will not be correctly calculated.

* Temporary DataFrame can be saved and used.

ex:- `d2 = data.loc[(data["Height"]>143)&(data["Gender"]=='m')]`

* If we want to get only one column based on these conditions.

i.e Name of People with 'Height' > 143 & 'Gender' = Male

```
d4 = data.loc[(data['Height']>143)|(data['Gender']=='m'),['Name']]
```

↑ Only 'Name' will be outputted.

d4

↘

	Name
0	P1
2	P2
3	P4
4	P5

* If we want to just see Name & Age

```
d5 = data.loc[(data['Height'] >143) | (data['Gender']
                =='m'), ['Name', 'Age']]
```

d5 ↘

	Name	Age
0	P1	23
2	P3	58
3	P4	24
4	P5	65

Based on the condition we can modify the values here based on this condition 'Name' is made 0.

```
data.loc[(data['Height'] >143) & (data['Gender']=='m'),
                ["Name"]] = 0
```

data ↘

	Name	Age	Gender	Height
0	0	23	m	153.2
1	P2	21	f	123.1
2	P3	58	f	173.8
3	P4	24	m	123.2
4	0	65	m	163.8
5	P6	12	f	121.1

If we want to assign values to more than one we need to give correct no of values

```
data.loc[(data['Height']>143) & (data['Gender']=='m'),
        ["Name","Age"]] = [0, 1]
```

data

	Name	Age	Gender	Height
0	0	1	m	153.2
1	P2	21	f	123.1
2	0	1	f	173.8
3	0	1	m	123.2
4	0	1	m	163.8
5	P6	12	f	121.1

* To find no.of rows or columns we can use shape.

```
data.shape
    → (6,4)
```

no.of rows — `data.shape[0]` → 6

no.of columns — `data.shape[1]` → 4

Assignment

On the given student Performance.csv

1) Create a column of percentage based on math, reading and writing score.
2) Take out male scored more than 50%.
3) " " " " less " ".
4) " " female " more " ".
5) " " " " less " ".
6) How many female candidates.
7) " " male " .
8) Students whose parents level education is Bachelor's degree.
9) How many males in standard lunch.
10) " " " " free lunch.
11) " " females in standard lunch.
12) " " " " free lunch.

* if we want to find unique values we can use unique( ) function.

ex:- data["lunch"].unique()

↳ array(['standard', 'free/reduced'], dtype = object)