Read in data and store it in a DataFrame

```
read_csv(filename, sep, index_col, names, header, skirows, parse_dates)
```

sep='character', defines the field separator. By default, is set to ','
sep='\s+' will set the field separator to whitespaces, including tabs

Select a field to label the rows
index_col=field number (starts from 0).
If not specified, rows are indexed with integer numbers 0 to n-1.

Specify column labels
names=list of values If not specified, column labels are taken from the first row of the file

Specify which row to be used to label the columns
header=0, the first row is considered as the header.
header = None will assign default integer numbers to the column labels.
header=number - will pick a row number to label the columns

Skip comment lines
comment='character'

Read in data and store it in a DataFrame

```
read_csv(filename, sep, index_col, names, header, skirows, parse_dates)
```

Skip comment lines
comment='character’

Skip rows
skiprows=int - will skip int number of lines
skiprows=list of numbers - will skip these row numbers

Read in specific fields
usecols=list of numbers, or list of names - to read in specific columns

Deadline with datetime data
parse_dates convert the specified columns, containing date or datetime-like strings, into datetime objects.
parse_dates=['Column1', 'Column2', ...]

Read in data and store it in a DataFrame - Examples

read_csv() uses a comma as the default separator.

sample-csv.csv

```
name,age,state,point
Alice,24,NY,64
Bob,42,CA,92
Charlie,18,CA,70
Dave,68,TX,70
Ellen,24,CA,88
Frank,30,NY,57
```

Here we just read the data set, without using other parameters

```
# column names are taken from the first row of the file
# row indices are set to 0 .. n-1
df1=pd.read_csv('sample-csv.csv')
```

```
      name  age state  point
0    Alice   24    NY     64
1      Bob   42    CA     92
2  Charlie   18    CA     70
3     Dave   68    TX     70
4    Ellen   24    CA     88
5    Frank   30    NY     57
```

Read in data and store it in a DataFrame - Examples

read_csv() uses a comma as the default separator.
sample-csv.csv

```
name,age,state,point
Alice,24,NY,64
Bob,42,CA,92
Charlie,18,CA,70
Dave,68,TX,70
Ellen,24,CA,88
Frank,30,NY,57
```

To label the rows with a specific field:

```
#index_col=0 will label the rows with the first field
df2=pd.read_csv('sample-csv.csv', index_col=0)
```

name	age	state	point
Alice	24	NY	64
Bob	42	CA	92
Charlie	18	CA	70
Dave	68	TX	70
Ellen	24	CA	88
Frank	30	NY	57

Read in data and store it in a DataFrame - Examples

To label the columns with a specific row:

```
# Label the columns with the second row
df3=pd.read_csv('sample-csv.csv', header=2)
```

To read in specific fields, and so skip others:

```
#read in 3rd and 4th fields
df4=pd.read_csv('sample-csv.csv', usecols=[2,3])
```

Read in data and store it in a DataFrame - Examples

sample-comments-sep.txt

```
#comment lines
#comment lines
name:age:state:point
Alice:24:NY:64
Bob:42:CA:92
Charlie:18:CA:70
Dave:68:TX:70
#comment lines
Ellen:24:CA:88
Frank:30:NY:57
```

Set the 3rd field to label the rows, and exclude comment lines

```
df5=pd.read_csv('sample-comments-sep.txt', sep=':',
       comment='#', index_col=2)
```

state	name	age	point
NY	Alice	24	64
CA	Bob	42	92
CA	Charlie	18	70
TX	Dave	68	70
CA	Ellen	24	88
NY	Frank	30	57

Read in data and store it in a DataFrame - Examples

Here is an example of a data set containing dates, sample-dates.csv

```
Name,Age,State,Score,Birthdate
Alice,24,NY,64,1999-05-15
Bob,42,CA,92,1981-02-28
Charlie,18,CA,70,2006-11-03
Dave,68,TX,70,1956-07-20
Ellen,24,CA,88,1999-08-12
Frank,30,NY,57,1993-10-05
```

```
df5=pd.read_csv(sample-date.csv, parse_dates=['Birthdate'])
```

Converting to a datetime is very useful when you want to plot the datetime values.
You can also explore the pd.to_datetime() function for converting fields to a datetime object.

To read in data pandas provides other functions to read excel, jason files etc.
More info here https://pandas.pydata.org/docs/user_guide/io.html

DataFrame operations useful for data analysis

Getting info about the data

info() provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

```
df.info()
```

Viewing the data

head() and tail() methods are bash-like commands.

By default, they output the first and last five rows of a DataFrame, but we could also pass a number.

```
df.head(10) #outputs the top ten rows
```

DataFrame operations useful for data analysis

Check and remove duplicate rows

Check for duplicate rows

```
df.duplicated() #returns a Boolean Series
```

To know how many duplicate roes there are we use the sum function, which will sum the True values:

```
df.duplicated().sum() #sum the True values
```

Remove duplicate rows

```
df.drop_duplicates(inplace=True)  #drops duplicate rows
```

DataFrame operations useful for data analysis

Detect and remove missing values

Missing values are represented in padas as NaN for numeric and string values, and with NaT for datetime values.

We use **isnull() or isna()** for detecting missing values:

```
df.isnull() #returns a Boolean DataFrame

df.isnull().sum() #total number of missing values (True) in each
                  column
```

You can remove missing values by using **dropna()**

```
df.dropna(inplace=True) #delete any row containing missing values
```

#you can also drop columns containing missing values by setting axis=1

```
df.dropna(axis=1, inplace=True)
```

Adding rows

To add a row in DataFrame, we can use the concat() function, which concatenates DataFarmes. This function is useful if you want to concatenate different data sets.

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

Create a DataFrame containing rows:

```
new_row = pd.DataFrame.from_dict({'LastName': ['Clara'],
                                  'Age': 40,
                                  'Height': 70.0})
```

Concatenate the two DataFrames

```
df = pd.concat([new_row, df], ignore_index=True)
```

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8
5	**Clara**	**40**	**70.0**	**NaN**

ignore_index=True is used to re-set the indices of the resulting dataframe

Add a column at the end using [ ]

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

We want to add the High Blood Pressure values of patients at the end the DataFrame df.

```
L=[124, 109, 125, 117, 122] # define a list of values
```

Add the list as a column, and name the column 'HighBP'

```
df["HighBP"]=L
```

	LastName	Age	Height	Weight	HighBP
0	Sanchez	38	71.2	176.1	124
1	Johnson	43	69.0	163.5	109
2	Zhang	38	64.5	131.6	125
3	Diaz	40	67.4	133.1	117
4	Brown	49	64.2	119.8	122

Add a column at the end using [ ]

You can also perform vectorized operations between columns (which are series type) , and add the result to a DataFrame

```
df["BMI"]=(df['Weight']*0.453592)/(df['Height']*0.0254)**2
```

	LastName	Age	Height	Weight	HighBP	BMI
0	Sanchez	38	71.2	176.1	124	24.422905
1	Johnson	43	69.0	163.5	109	24.144462
2	Zhang	38	64.5	131.6	125	22.239981
3	Diaz	40	67.4	133.1	117	20.599478
4	Brown	49	64.2	119.8	122	20.435474

Add a column at a specific position using insert()

Dataframe.insert() is used to insert a column to a Dataframe at a specified index position. It is like the list insert method, and it updates the original DataFrame.

The general syntax is:

```
df.insert(index_position, column_name, value)
```

For example, we want to add Low Blood Pressure values after the HighBP column. We want to name the column LowBP, and insert the column at index 5, which is the 6th column.

```
Lv=[60, 75, 67, 85, 90, 82] #define a list of values
df.insert(5,'LowBP',Lv)
```

	LastName	Age	Height	Weight	HighBP	LowBP	BMI
0	Sanchez	38	71.2	176.1	124	60	24.422905
1	Johnson	43	69.0	163.5	109	75	24.144462
2	Zhang	38	64.5	131.6	125	67	22.239981
3	Diaz	40	67.4	133.1	117	85	20.599478
4	Brown	49	64.2	119.8	122	90	20.435474

Remove rows and columns - drop() method

To delete columns and rows of DataFrame, we can use the **drop()** method. The drop() method by default returns a new DataFrame with the modified values. If you want to modify the same dataFrame, use inplace=True

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

To delete columns, use column labels, and axis=1

```
df.drop(['Height','Weight'],axis=1) #delete columns
```

To delete rows, use row labels, and axis=0, which is default

```
df.drop([0,3]) #delete rows
```

You can also use the inplace=True

Sorting

To sort you can use:

- sort_values() to sort values (the data) along an axis
- sort_index() to sort labels along an axis

axis=0 (default) row-wise, axis=1 column-wise

	LastName	Age	Height	Weight	BMI
4	Brown	49	64.2	119.8	20.435474
1	Johnson	43	69.0	163.5	24.144462
3	Diaz	40	67.4	133.1	20.599478
0	Sanchez	38	71.2	176.1	24.422905
2	Zhang	38	64.5	131.6	22.239981

We want to sort values row-wise by a column. Default is ascending order.

```
df.sort_values('Age', ascending=False) # sort values by Age
```

	LastName	Age	Height	Weight	BMI
4	Brown	49	64.2	119.8	20.435474
1	Johnson	43	69.0	163.5	24.144462
3	Diaz	40	67.4	133.1	20.599478
0	Sanchez	38	71.2	176.1	24.422905
2	Zhang	38	64.5	131.6	22.239981

Explore the parameters ignore_index of the sort_values().

Sorting

To sort you can use:

- sort_values() to sort values (the data) along an axis
- sort_index() to sort labels along an axis

axis=0 (default) row-wise, axis=1 column-wise

To sort by row labels:

```
df.sort_index())
```

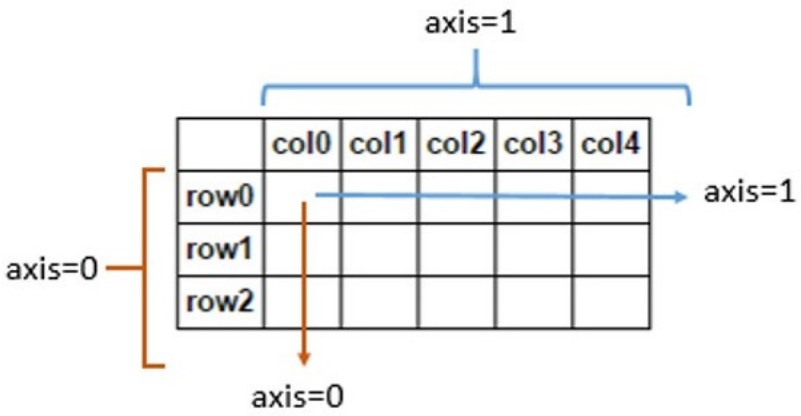
LastName	Age	Height	Weight	BMI
Sanchez	38	71.2	176.1	24.422905
Johnson	43	69.0	163.5	24.144462
Zhang	38	64.5	131.6	22.239981
Diaz	40	67.4	133.1	20.599478
Brown	49	64.2	119.8	20.435474

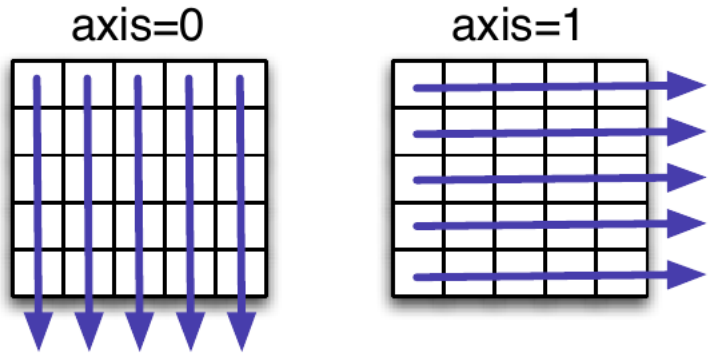
To sort by column labels:

```
df.sort_index(axis=1))
```

LastName	Age	BMI	Height	Weight
Sanchez	38	24.422905	71.2	176.1
Johnson	43	24.144462	69.0	163.5
Zhang	38	22.239981	64.5	131.6
Diaz	40	20.599478	67.4	133.1
Brown	49	20.435474	64.2	119.8

Statistics methods





axis=0 (default) means operations are performed row-wise, i.e., along the vertical axis.
axis=1 means operations are performed column-wise, i.e., along the horizontal axis.

```
count()   - number of non-NA observations
sum()     - sum of values
mean()    - mean of values
min()     - minimum
max()     - maximum
abs()     - absolute Value
prod()    - product of values
std()     - standard deviation
cumsum()  - cumulative sum
cumprod()- cumulative product
idxmin()  - index of the minimum
idxmax()  - index od the maximum
```

Statistics methods - Examples

```
m=df['Age'].min() #minimum of a Series

idm=df['Age'].idxmin() # index of the minimum of a Series


mv=df.min(axis=0) #minimum down (vertically)


mh=df.min(axis=1) #minimum across (horizontally)
```

Generate summary of statistics

The **describe()** method is very useful because returns a summary statistics of a DataFrame for each column.

	LastName	Age	Height	Weight	BMI
0	Sanchez	38	71.2	176.1	24.422905
1	Johnson	43	69.0	163.5	24.144462
2	Zhang	38	64.5	131.6	22.239981
3	Diaz	40	67.4	133.1	20.599478
4	Brown	49	64.2	119.8	20.435474

```
df.describe() #returns a DataFrame
```

	Age	Height	Weight	BMI
count	5.000000	5.000000	5.000000	5.000000
mean	41.600000	67.260000	144.820000	22.368460
std	4.615192	2.981275	23.798676	1.887933
min	38.000000	64.200000	119.800000	20.435474
25%	38.000000	64.500000	131.600000	20.599478
50%	40.000000	67.400000	133.100000	22.239981
75%	43.000000	69.000000	163.500000	24.144462
max	49.000000	71.200000	176.100000	24.422905

Writing data

You can write data in csv format by using the **to_csv** function
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html

Looping

```
for col in df:  #loop over column labels
     print(col)


for r in df.index.values:  #loop over row labels
     print(r)
```

You can also use df.**iterrows()** to iterate over DataFrame rows as (index, Series) pairs.