pandas

pandas (*panel data)* is a Python library designed for working with tabular data.

Built on NumPy's foundation, Pandas inherits and extends many of NumPy's array-based features.

Pandas is best used for working with heterogeneous and labeled data
NumPy is best used for working with homogeneous numerical arrays

If you do not have it, you can install pandas

```
conda install pandas  #if you have ananconda
pip3 install pandas   #if you do not have ananconda
```

To import Pandas

```
import pandas as pd
```

Pandas Object: Series

Characteristics and usage

- is a labeled 1D array - it is an analog of a 1D NumPy array with labeled indices.
- Homogeneous
- Mutable
- Fixed size
- Each element in a Series is associated with an index (label), which can be customized or automatically generated.

Usage of Series: Series are utilized to represent labeled data.
Example: storing student ages for an online course.
Series allows customization of indices (labels, depicted in blue), which means you can assign student names as labels, providing a more intuitive way to access each student's age directly by name.

```
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
```

Creating Series Objects


```
pd.Series(data=None, index=None)
```

parameters:

data: list, dictionary, ndarray, a scalar value
index: optional parameter, used to customize the indices.

Create a Series from a list

If the index parameter is not specified, labels are automatically generated to be integer numbers 0,1,2,.. like Python-style indices

```
Age= [38, 43, 38, 40, 49]
s=pd.Series(Age)
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64
```

dtype: int64 is the dtype of the values

Create a Series from a list: customize the labels

index parameter: We can customize the indices by setting the index parameter to a list

```
Age = [38, 43, 38, 40, 49]
LastName = ["Sanchez", "Johnson", "Zhang", "Diaz", "Brown"]
sl=pd.Series(Age, index=LastName)
print(sl)
```

```
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

We can also label the indices with nonsequential numbers:

```
sll=pd.Series(Age, index=[1,10,34,56,70])
print(sll)
```

```
1     38
10    43
34    38
56    40
70    49
dtype: int64
```

Create a Series from a dictionary

A Series can be created out of a dictionary,
in which case the indices default to the dictionary keys:

```
D={'Sanchez': 38, 'Johnson': 43, 'Zhang': 38,
   'Diaz': 40, 'Brown': 49}

sd=pd.Series(D)
print(sd)
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

A series can also be seen as dictionary-like, where each value has an associated label

Create a Series from a dictionary: index parameter

index parameter: In the case of a dictionary, the index parameter can be explicitly set to control the order and/or the subset of keys used.

```
D={'Sanchez': 38, 'Johnson': 43, 'Zhang': 38, 'Diaz': 40, 'Brown': 49}
L=['Brown', 'Diaz', 'Johnson', 'Sanchez', 'Zhang'] # list of keys sorted
```

If we set the index to L, the order of the elements in the Series follows the elements in the list:

```
sd=pd.Series(D, index =L)
print(sd)
```

```
Brown      49
Diaz       40
Johnson    43
Sanchez    38
Zhang      38
dtype: int64
```

We can also make a Series from a subset of key: value pairs in the order we decide:

```
sd=pd.Series(D, index =['Diaz', 'Johnson'])
print(sd)
```

```
Diaz       40
Johnson    43
dtype: int64
```

Attributes of a Series

A Series is like a 1D array, and it has similar attributes:

```
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64

s.ndim  #1
s.shape #(5,)
s.size  #5

s.values #returns a 1D array of the values
[38 43 38 40 49]

s.index  #returns the Index Object of the labels
RangeIndex(start=0, stop=5, step=1)
```

means a range of integers in range [0 ,5) with a step of 1.

```
s.index.values #returns the labels as 1D array
[0 1 2 3 4]
```

Attributes of a Series

```
print(sd)
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

```
sd.index
Index(['Sanchez', 'Johnson', 'Zhang', 'Diaz', 'Brown'],
dtype='object')
```

dtype='object' specifies the data type of the elements. Here, 'object' typically denotes strings in Pandas.

```
sd.index.values
['Sanchez' 'Johnson' 'Zhang' 'Diaz' 'Brown'] #1D array of strings.
```

In pandas the type of the labels is an Index Object

Series Type Conversion

Converting a Series to a 1D NumPy array

```
np.array(sd)
sd.values)
sd.to_numpy()
[38 43 38 40 49]
```

```
print(sd)
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

Convert a Series to a list type

```
list(sd)
sd.to_list()
[38, 43, 38, 40, 49]
```

Converting a Series to a dictionary type

```
dict(zip(sd.index, sd.values))
sd.to_dict()
{'Sanchez': 38, 'Johnson': 43, 'Zhang': 38, 'Diaz': 40, 'Brown': 49}
```

DataFrame Object

Characteristics and usage

- is a 2D labeled tabular structure, and it is an analog of a 2D NumPy array with labelled rows and columns (depicted in blue)
- Heterogeneous
- Mutable
- Size can change
- is a collection of Series
- each element in a DataFrame is associated with row and column indices (labels), which can be customized or automatically generated.

Usage: DataFrames are ideal for handling heterogeneous data with labeled rows and columns and for representing tabular data: rows correspond to instances (examples, observations, etc.), and columns correspond to features of these instances.

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

Creating DataFrame Objects

To create a DataFrame out of other python Objects, we will use the pd.DataFrame() constructor.

```
pd.DataFrame(data=None, index=None, columns=None)
```

data: dictionary can contain ndarray, lists, Series
- 2D ndarray
- Series
- Pandas DataFrame

index: sets the row labels. Default row indices are 0,1,2..
columns: sets the column labels. The default column indices are 0,1,2..

Create a DataFrame from a 2D array


```
arr1=np.array([[1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25, 125]])
dfnp = pd.DataFrame(arr1) #if index and column parameters are missing, indices automatically default to python-style 0,1,2,
print(dfnp)
```

	0	1	2
0	1	1	1
1	2	4	8
2	3	9	27
3	4	16	64
4	5	25	125

To customize the row and column labels we set the index and columns parameters.

```
index_val=['first','second','third','fourth','fifth'] #list of row labels
column_val=['number', 'squares', 'cubes']   #list of column labels
dfnp = pd.DataFrame(arr1, index=ndex_val, columns=column_val)
print(dfnp)
```

	number	squares	cubes
first	1	1	1
second	2	4	8
third	3	9	27
fourth	4	16	64
fifth	5	25	125

Create a DataFrame from a dictionary of lists


```
D={"LastName": ["Sanchez", "Johnson", "Zhang", "Diaz", "Brown"],
"Age": [38, 43, 38, 40, 49], "Height": [71.2, 69.0, 64.5, 67.4, 64.2],
"Weight": [176.1, 163.5, 131.6, 133.1, 119.8]}

df=pd.DataFrame(D)
print(df)
```

	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

The dictionary keys will be used as column labels and the values in each list as the values (data) in the columns of the DataFrame.

If we do not use the index parameter, pandas automatically generates the row labels, which default to the the normal Python indices 0,1,2,..

Create a DataFrame from a dictionary of lists – customize row labels

We can **customize the row** labels by defining them via the **index parameter** of the pd.DataFrame()

```
 D={"LastName": ["Sanchez", "Johnson", "Zhang", "Diaz", "Brown"],
"Age": [38, 43, 38, 40, 49], "Height": [71.2, 69.0, 64.5, 67.4, 64.2],
"Weight": [176.1, 163.5, 131.6, 133.1, 119.8]}

dfc=pd.DataFrame(D, index=['A','B','C','D','E'])
print(dfc)
```

```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

Changing row labels and column labels of an existing DataFrame

We can change row and column labels using attributes:

```
df.columns = new_columns
df.index = new_index
```

```
print(dfnp)
```

	number	squares	cubes
first	1	1	1
second	2	4	8
third	3	9	27
fourth	4	16	64
fifth	5	25	125

To change the row labels:

```
dfnp.index=[10,20,30,40,50]
print(dfnp)
```

	number	squares	cubes
10	1	1	1
20	2	4	8
30	3	9	27
40	4	16	64
50	5	25	125

To change the column labels:

```
dfnp.columns=['A', 'B','C']
print(dfnp)
```

	A	B	C
10	1	1	1
20	2	4	8
30	3	9	27
40	4	16	64
50	5	25	125

Changing row labels of an existing DataFrame

The set_index() method is used to set the row labels using existing columns

```
print(dfnp)
 A   B    C
10   1    1
20   2    4
30   3    9
40   4   16
50   5   25
```

```
125
```

df1=dfnp.set_index('C') #we set the existing column "C" as row labels

```
     A   B
C
1    1   1
8    2   4
27   3   9
64   4  16
125  5  25
```

The set_index() method returns a new object which is a copy of the original DataFrame object. If you want the original DataFrame object to be modified, you can use the parameter **inplace=True**.

```
dfnp.set_index('B', inplace=True)  #will modify directly dfnp
```

DataFrame - the info() method

The info() method in Pandas provides a concise summary of a DataFrame, including information about the index, columns, data types, non-null values, and memory usage.

print(df)

```
  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
```

print(df.info())

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   LastName  5 non-null      object
 1   Age       5 non-null      int64
 2   Height    5 non-null      float64
 3   Weight    5 non-null      float64
dtypes: float64(2), int64(1), object(1)
memory usage: 288.0+ bytes
None
```

DataFrame - attributes

NumPy-like attributes

```
df.size
df.shape
df.ndim
```

Additional attributes

```
df.index          #returns Index Object of row labels
df.columns        #returns Index Object of column labels

df.index.values    #returns 1D array of row labels
df.columns.values #returns 1D array of column labels


df.index.dtype      #returns the type of row labels
df.columns.dtype    #return the type of the column labels
```

Data Indexing and Selection

Indexers: loc [] and iloc[]

An indexer is a mechanism that allows users to locate and access specific subsets of data, including rows, columns, or individual elements.

Pandas main two Indexers are:

.loc[] allows indexing methods that always reference the labels of rows and columns, the visible ones, no matter if they are the defaults (0,1,2, etc) of customized.
Use: convenient when you want to select data using row and column labels.

.iloc[] allows indexing methods that always reference the Python-style indices based on position (0,1,2, .. no matter if they are visible or not).
Use: convenient choice when labels are not significant or when you want to perform operations based on the numerical position of data. This is usually faster than loc, and maintains consistency with the use integer-based indexing of NumPy arrays.

The use of loc[] and iloc[] can prevent subtle bugs due to the mixed indexing/slicing convention. Alo iloc

Indexing a Series

A Series can be indexed by using loc[] or iloc[]:

```
.iloc[] #use positional indices – usual python/NumPy indexing syntax

iloc[index]                          Single element Indexing
iloc[[index1,index2, index3]]        Array Indexing (Fancy indexing)
iloc[ index_start : index_end]       Slicing, the start is included, end excluded
iloc[boolean array]                  Boolean Indexing (Masking)
```

```
.loc[] #use the labels (defaults or customized)

loc[label]                           Single element Indexing
loc[[ label1, label2, label3 ]]      Array Indexing (Fancy indexing)
loc[ label_start: label_end ]        Slicing - both the start and end included
loc[boolean array/series]            Boolean Indexing (Masking)
```

Indexing a Series – iloc[ ]

When using iloc[] we index a Series as you would index a 1D array (no matter what the labels are)

```
print(sl)
Sanchez     38
Johnson     43
Zhang       38
Diaz        40
Brown       49
dtype: int64
```

```
print(sl.iloc[1])  #single element indexing
43

print(sl.iloc[ [1,3]]) #array indexing
Johnson     43
Diaz        40
dtype: int64
```

To use Boolean indexing with iloc[], we convert the Boolean Series to a Boolean array. Iloc[] does not support a Boolean series, while loc[] does.

```
bool1= sl > 30) & (sl < 50) #creates a Boolean Series
arr1= np.array(bool1)
print(sl.iloc[ arr1] )
Johnson     43
Brown       49
dtype: int64
```

Indexing a Series – iloc[ ]

To index a Series with iloc[] we use the same Python syntax we would use to index a 1D array (no matter what the labels are). iloc[]. uses positional indices.

```
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64
```

```
s.iloc[0] #access first element
38
```

```
s.iloc[[1,3]] #indexing with a list of indices
1    43
3    40
dtype: int64
```

```
s.iloc[:2] #slice from index 0 to index 1
0    38
1    43
dtype: int64
```

Indexing a Series – loc[ ]

To index a Series with loc[] we use the labels.
To obtain a 1D array of the labels:

```
print(sl)
Sanchez     38
Johnson     43
Zhang       38
Diaz        40
Brown       49
dtype: int64
```

```
print(sl.index.values)
['Sanchez' 'Johnson' 'Zhang' 'Diaz' 'Brown']
```

```
sl.loc['Sanchez'] #access the value labeled with 'Sanchez'
38
```

```
sl.loc[:'Zhang']  #slice to the element labeled with 'Zhang'
Sanchez     38
Johnson     43
Zhang       38
dtype: int64
```

```
sl.loc[['Sanchez','Zhang']] #select multiple elements with list of labels
Sanchez     38
Zhang       38
dtype: int64
```

```
bool2=(sl > 40) & (sl < 50) #masking with Boolean Series
sl.loc[bool2]
Johnson     43
Brown       49
dtype: int64
```

Indexing a Series – loc[ ]

Now we use loc[] on this Series, where the indices default to integer numbers 0, 1, 2, .. .
They are the row labels in this case.

To obtain a 1D array of row labels:

```
print(s.index.values)
[0 1 2 3 4]
```

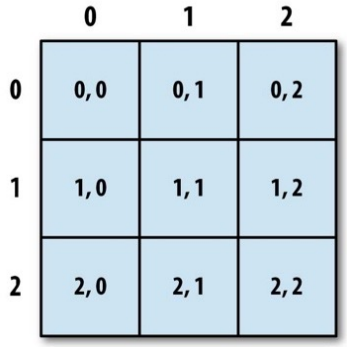
```
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64
```

Now we use those labels to index with loc[]

```
print(s.loc[:2]) #notice when slicing with loc the end label is inlcuded
0    38
1    43
2    38
dtype: int64dtype: int64
```

Indexing a DataFrame with iloc[]

.iloc[] integer-Based Indexing (like indexing a 2D array): with .iloc, you can select rows and columns based solely on their integer positions, regardless of the row and column labels



Single Label Indexing:

df.iloc[row_index] Accesses the row at index row_index.

df.iloc[:, column_index] Accesses the column at index column_index.

Slicing with Indices:

df.iloc[start_index :end_index] Accesses rows from start_index to end_index-1.

df.iloc[:, start_index:end_index] Accesses columns from start_index to end_index-

Array Indexing with Indices:

df.iloc[[index1, index2, index3]] Accesses rows at indices index1, index2, and index3.

df.iloc[:, [index1, index2, index3]] Accesses columns at indices index1, index2, and index3.

Combination of Indices and Slicing:

df.iloc[row_index, column_index] Accesses the element at row row_index, column column_index.

df.iloc[start_row:end_row, start_column:end_column] Accesses a subset of rows and columns.

Indexing a DataFrame with iloc[]

Use same syntax you would use to index a 2D array

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

df.iloc[::2] # slice every two rows

LastName	Age	Height	Weight
A Sanchez	38	71.2	176.1
C Zhang	38	64.5	131.6
E Brown	49	64.2	119.8

df.iloc[[1,3]] # list of indices [1,3] to select 2nd and 4th row

LastName	Age	Height	Weight
B Johnson	43	69.0	163.5
D Diaz	40	67.4	133.1

df.iloc[1,3] # select one element at row index 1, column index 3

163.5

Indexing a DataFrame with iloc[]

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

```
df.iloc[2] # same as df.iloc[2,:] select 3rd row, returns a series
LastName    Zhang
Age            38
Height       64.5
Weight      131.6
Name: 2, dtype: object
```

```
df.iloc[[2]] # returns a dataframe
```

	LastName	Age	Height	Weight
C	Zhang	38	64.5	131.6

Indexing a DataFrame with iloc[]

df.iloc[:,3] # select 4th column

```
A    176.1
B    163.5
C    131.6
D    133.1
E    119.8
Name: Weight, dtype: float64
```

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

df.iloc[:,[1,3]] # select 2nd and 4th columns

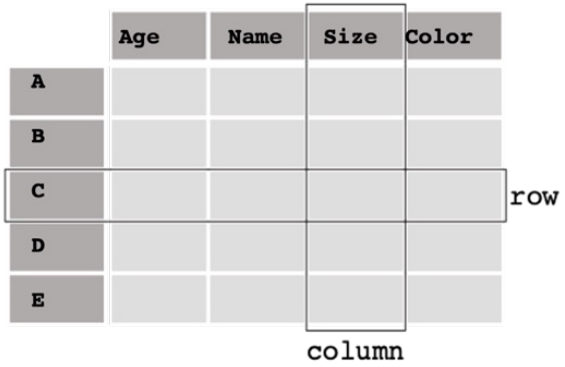
	Age	Weight
A	38	176.1
B	43	163.5
C	38	131.6
D	40	133.1
E	49	119.8

df.iloc[[1,3],[1,3]] # select 2nd and 4th rows, 2nd and 4th columns

	Age	Weight
A	43	163.5
D	40	133.1

Indexing a DataFrame with loc[ ]

The loc[] provides **label-based indexing**, When using .loc, you can specify rows and columns based on their labels, regardless of whether they are the default integer indices or custom index labels.



Single Label Indexing:

df.loc[row_labe'] Accesses the row with label row_label.

df.loc[:, column_label] Accesses the column with label column_label.

Slicing with Labels:

df.loc[start_label : end_label] Accesses rows from start_label to end_label (inclusive).

df.loc[:, start_label : end_label] Accesses columns from start_label to end_label (inclusive).

Array Indexing:

df.loc[[label1, label2, label3]] Accesses rows with labels label1, label2, and label3.

df.loc[:, ['label1, label2, label3]] Accesses columns with labels label1, label2, and label3

Combination of Labels and Slicing:

df.loc[row_label, column_label] Accesses the element at row row_label, column column_label.

df.loc[start_row : end_row, start_column : end_column] Accesses a subset of rows and columns.

Indexing a DataFrame with loc[ ]

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

When using loc[] use the labels

```
df.columns.values #return 1D array of the row labels
['LastName', 'Age', 'Height', 'Weight']

df.index,values #return 1D array of the column labels
['A', 'B', 'C', 'D', 'E']
```

```
df.loc['A','Height'] # select one element at row label 'A' and
columns label 'Height'
```

Indexing a DataFrame with loc[ ]


```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

```
dfc.loc['A'] # selects row labeled 'A', and returns it as Series
LastName    Sanchez
Age              38
Height         71.2
Weight        176.1
Name: A, dtype: object
```

If you want to select and return a row as a DataFrame, you should pass a list:

```
dfc.loc[['A']] #a list of labels returns a DataFrame
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
```

Indexing a DataFrame with loc[ ]

Use labels

```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

df.loc[:,'Age':] # select from column 'Age' to the end

```
   Age  Height  Weight
A   38    71.2   176.1
B   43    69.0   163.5
C   38    64.5   131.6
D   40    67.4   133.1
E   49    64.2   119.8
```

df.loc[['A','D'],'Age':] #select rows 'A' and 'D' and columns 'Age' to the end

```
   Age  Height  Weight
A   38    71.2   176.1
D   40    67.4   133.1
```

Indexing a DataFrame with loc[ ]


```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

```
df.loc[['D','A'],['Age','Height']]
```

```
    Age  Height
D    40    67.4
A    38    71.2
```

```
df.loc['D':] # select rows from 'D' to the end
```

```
df.loc['D'] # select row D
```

Other ways to select columns: use [ ] or dot notation


```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

You can use [ ]

```
df['Age'] # select column 'Age' and returns a series
```

```
df[['Age']] # select column 'Age' and returns a dataframe
```

```
df[['LastName','Height']] # select columns 'LastName' and 'Height'
```

Or you can use the dot notation, also called attribute access

```
df.Age
0    38
1    43
2    38
3    40
4    49
Name: Age, dtype: int64
```

Boolean Indexing

We extract information based on conditions.

```
comparison operators:
==  equals
!= not equals
> greater than
< less than
>= greater than or equal to
<= less than or equal to

Logical operator - each condition must be put in a
separate pair of brackets.
  & (and)
  | (or
  ~ (not)
```

Boolean Indexing with loc[]

loc[] works with label and also with Boolean Series, Boolean array, and a list of Boolean values.

The index boolean can be: Boolean Series, a list of Boolean, and ndarray of Boolean

Boolean Indexing with Rows:

df.loc[boolean] Selects rows where the corresponding value in boolean is True.

Boolean Indexing with Columns:

df.loc[:, boolean] Selects columns where the corresponding value in Boolean is True.

Combining Boolean Indexing:

df.loc[boolean, column_label] Selects True rows and the column specified by column_label

df.loc[boolean, [label1, label2, label3]] Selects True rows and specific columns

df.loc[boolean, start_column : end_column] Selects True rows and columns from start_column to end_column (inclusive).

Boolean Indexing – use loc[]


```
#notice each condition is between parenthesis
bool2=(dfc['Age'] >= 40) & (dfc['Height'] < 70)

df.loc[bool2] # extract rows satisfying the condition
```

```
  LastName  Age  Height  Weight
B  Johnson   43    69.0   163.5
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

```
df.loc[bool2,'LastName'] # extract LastName column of the rows satisfying the condition
```

```
B    Johnson
D       Diaz
E      Brown
Name: LastName, dtype: object
```

```
df.loc[bool2,['Age','LastName']]
```

```
   Age LastName
B   43  Johnson
D   40     Diaz
E   49    Brown
```

Boolean Indexing – use iloc[]

iloc[] works with integer indices (like NumPy) and with a Boolean array or a list of Boolean values. Does not work with Boolean Series.

Boolean Indexing with Rows:
df.iloc[boolean] Selects rows where the corresponding value in boolean is True.

Boolean Indexing with Columns:
df.iloc[:, boolean] Selects columns where the corresponding value in boolean is True.

Combining Boolean Indexing:
df.iloc[boolean, column_index] Selects True rows and the column at column_index.

df.iloc[boolean, start_column:end_column] Selects True rows and columns from start_column to end_column (inclusive).

df.loc[boolean, [column1, column2, colum3]] Selects True rows specific columns

If you use iloc[] you should convert the Boolean Series to a numpy array or list
`df.iloc[np.array(bool1)])`