Data Indexing and Selection

In [ ]:

```
import pandas as pd
import numpy as np
```

In [ ]:

```
data_frame = pd.read_csv("students.csv")
data_frame
```

Viewing index and columns names

In [ ]:

```
data_frame.index
```

In [ ]:

```
data_frame.columns
```

In [ ]:

```
data_frame.dtypes
```

Viewing Data

head() helps to view topn rows. Default is 5 rows.

In [ ]:

```
data_frame.head()
```

In [ ]:

```
#top n rows can be see by passing the topn value
data_frame.head(2)
```

tail() helps to view bottom n rows. Default is 5 rows.

In [ ]:

```
data_frame.tail()
```

In [ ]:

```
data_frame.tail(2)
```

Column Selection

Accees individual columns by its name. i.e. data_frame_obj.column_name

In [ ]:

```
data_frame.id # returns index and data values
```

Basically its a series and then all series operations can be performed on it.

In [ ]:

```
type(data_frame.id)
```

[] operator can also be used to access the column. The result is also a series.

In [ ]:

```
data_frame["name"]
```

In [ ]:

```
type(data_frame["name"])
```

List of columns can be accessed as follows :

In [ ]:

```
list_of_columns= ["id", "name"]
data_frame[list_of_columns]
```

But this time, the outcome is data frame not a series.

In [ ]:

```
type(data_frame[list_of_columns])
```

In [ ]:

```
sub_df = data_frame[list_of_columns]
sub_df
```

Different Choices for Indexing

Object selection has had a number of user-requested additions in order to support more explicit location based indexing.

.loc is primarily label based, but may also be used with a boolean array. .loc will raise KeyError when the items are not found. Allowed inputs are:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index.).
- A list or array of labels ['a', 'b', 'c'].
- A slice object with labels 'a':'f' (Note that contrary to usual python slices, both the start and the stop are included, when present in the index! See Slicing with labels.).
- A boolean array

.iloc is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array. .iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy slice semantics). Allowed inputs are:

- An integer e.g. 5.
- A list or array of integers [4, 3, 0].
- A slice object with ints 1:7.
- A boolean array.

Using .loc

In [ ]:

```
data_frame
```

Access the first row in the data frame using the index

In [ ]:

```
data_frame.loc[0]
```

Access the intermediate rows in the data frame using the index

In [ ]:

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

With index, column names can also be specified in order to access value for those columns only.

In [ ]:

```
data_frame.loc[0, "id"]
```

In [ ]:

```
data_frame.loc[0, ['id', 'name', 'quiz1']]
```

Other variations

In [ ]:

```
#Get all the row after index 2
data_frame.loc[2:]
```

In [ ]:

```
#Get all the rows having index less than 3
data_frame.loc[: 3]
```

In [ ]:

```
#Get alternate rows
data_frame.loc[::2]
```

In [ ]:

```
data_frame.loc[0, 'name']
```

In [ ]:

```
data_frame.loc[0:2, ['id', 'name', 'quiz1']]
```

Using .iloc

Integer indices are only used to access the values.

In [ ]:

```
data_frame
```

In [ ]:

```
#Access the zeroth row
data_frame.iloc[0]
```

In [ ]:

```
#Access first three rows
data_frame.iloc[0:3]
```

In [ ]:

```
#Acess every alternate row
data_frame.iloc[::2]
```

In [ ]:

```
#Acces zeroth row , zeroth column
data_frame.iloc[0, 0]
```

In [ ]:

```
#Aceess first two columns of row with index 1 i.e. second row
data_frame.iloc[1, 0:2]
```

In [ ]:

```
#Access explicit columns of second row
data_frame.iloc[1, [0, 2, 3]]
```

Filtering

Filter conditions can be used in [] to select the few specific rows.

In [ ]:

```
data_frame
```

In [ ]:

```
#Access the rows which fulfills the condition
data_frame[ data_frame.quiz1 > 3]
```

In [ ]:

```
#Access the rows which fulfills both condition
data_frame[ (data_frame.quiz1 > 3) & (data_frame.quiz2 > 3)]
```

In [ ]:

```
#Access the rows which fulfills either of condition
data_frame[ (data_frame.quiz1 > 1) | (data_frame.quiz2 > 3)]
```

In [ ]:

```
#Access the rows which fulfills both condition (with negation)
data_frame[ ~(data_frame.quiz1 > 4)]
```

Using isin

In [ ]:

```
data_frame
```

DataFrame also has an isin() method. When calling isin, pass a set of values as either an array or dict. If values is an array, isin returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

In [ ]:

```
marks = [3, 4]
data_frame.isin(marks)
```

The query() Method

DataFrame objects have a query() method that allows selection using an expression. You can get the value of

the frame where column b has values between the values of columns a and c.

In [ ]:

```
data_frame.query('midsem > compre')
```

In [ ]:

```
data_frame.query('midsem > compre & quiz1 == quiz2')
```