Data Indexing and Selection

In [ ]:
```
# What is data indexing and selection?
# Accessing and modifying values in Pandas Series and DataFrame objects
```

Data Selection in Series

In [ ]:
```
# Pandas Series can behave like both onedimensional NumPy array, and in many ways a standard Python dictionary.
```

How to access the data when Series acts as dictionary?

In [1]:
```
# Let's import the pandas as pd and create some data

import pandas as pd
data = pd.Series([10, 20, 30, 40, 50], index = ['A', 'B', 'C', 'D', 'E'])
data
```

Out[1]:
```
A    10
B    20
C    30
D    40
E    50
dtype: int64
```

In [2]:
```
# We can use index method to access the values

data['A']
```

Out[2]:
```
10
```

In [ ]:
```
# We can also use dictionary-like Python expressions and methods to examine the keys/indices and values
```

In [3]:
```
'A' in data
```

Out[3]:
```
True
```

In [6]:
```
# In pandas series dictionary keys act as index objects
pd.Series.keys?
```

In [7]:
```
data.keys()
```

Out[7]:
```
Index(['A', 'B', 'C', 'D', 'E'], dtype='object')
```

In [8]:
```
pd.Series.items?
```

In [9]:
```
print(data.items()); print(list(data.items()))
```

```
<zip object at 0x066C2FD0>
[('A', 10), ('B', 20), ('C', 30), ('D', 40), ('E', 50)]
```

In [10]:
```
data['A'] = 45
data
```

Out[10]:
```
A    45
B    20
C    30
D    40
E    50
dtype: int64
```

How to access the data when Series acts as one-dimensional array?

In [11]:
```
# Slicing by explicit index

print(data)
print(data['A':'D'])
```

```
A    45
B    20
C    30
D    40
E    50
dtype: int64
A    45
B    20
C    30
D    40
dtype: int64
```

In [12]:
```
# Slicing by implicit integer index

print(data)
print(data[0:4])
```

```
A    45
B    20
C    30
D    40
E    50
dtype: int64
A    45
B    20
C    30
D    40
dtype: int64
```

In [13]:
```
# masking
data[(data > 20) & (data < 50)]
```

Out[13]:
```
A    45
C    30
D    40
dtype: int64
```

In [14]:
```
# fancy indexing
data[['A', 'E']]
```

Out[14]:
```
A    45
E    50
dtype: int64
```

Indexers: loc, iloc, and ix

In [15]:
```
# Let's print the data and use explicit and implicit index to access the data

data
```

Out[15]:
```
A    45
B    20
C    30
D    40
E    50
dtype: int64
```

In [16]:
```
# Python index starts from zero by default
# Implicit index :  final index is excluded in the slice

data[1:3]
```

Out[16]:
```
B    20
C    30
dtype: int64
```

In [17]:
```
# Explicit index : final index is included in the slice

data['B':'D']
```

Out[17]:
```
B    20
C    30
D    40
dtype: int64
```

In [ ]:
```
# When the data has integer index then it becomes bit confused when using implicit and explicit index
# Hence loc, iloc and ix are used
# Python also states that 'explicit' is always better than 'implicit'
```

In [18]:
```
# Let's create a data with explicit integer index to understand this concept

idata = pd.Series(['a', 'b', 'c', 'd', 'e', 'f'], index=[1, 3, 4, 5, 6, 7])
idata
```

Out[18]:
```
1    a
3    b
4    c
5    d
6    e
7    f
dtype: object
```

In [19]:
```
# explicit index when indexing
idata[1]
```

Out[19]:
```
'a'
```

In [20]:
```
# implicit index when slicing
idata[1:4]
```

Out[20]:
```
3    b
4    c
5    d
dtype: object
```

In [21]:
```
# The 'loc' attribute allows indexing and slicing that always references the explicit index:
# Let's see the signature of 'loc'
pd.Series.loc?
```

In [22]:
```
idata.loc[1]
```

Out[22]:
```
'a'
```

In [23]:
```
idata.loc[1:4]
```

Out[23]:
```
1    a
3    b
4    c
dtype: object
```

In [24]:
```
idata.loc[1:6]
```

Out[24]:
```
1    a
3    b
4    c
5    d
6    e
dtype: object
```

In [25]:
```
print(idata)
print(idata[3:6]) # implicit index without loc : also final index is excluded
print(idata.loc[3:6])  # explicit index with loc : also final index is included
```

```
1    a
3    b
4    c
5    d
6    e
7    f
dtype: object
5    d
6    e
7    f
dtype: object
3    b
4    c
5    d
6    e
dtype: object
```

In [26]:
```
# Let's see the signature of 'iloc'
pd.Series.iloc?
```

In [27]:
```
# The iloc attribute on the other hand allows indexing and slicing that always references the implicit Python-style in
dex:
# so iloc preserves implicit index notation
print(idata)
print(idata.iloc[1])# implicit
print()
print(idata[1])       # this is explicit index which is equivalent to idata.loc[1]
print()
print(idata.loc[1])  # explicit
```

```
1    a
3    b
4    c
5    d
6    e
7    f
dtype: object
b

a

a
```

In [ ]:
```
idata.iloc[1:4]
```

In [28]:
```
print(idata)
print(idata[3:6])       # implicit index without loc : also final index is excluded
print(idata.iloc[3:6])  # iloc works like implicit index so iloc is meant for implicit index preservation
```

```
1    a
3    b
4    c
5    d
6    e
7    f
dtype: object
5    d
6    e
7    f
dtype: object
5    d
6    e
7    f
dtype: object
```

In [ ]:
```
# Without loc
# Make a Note: --> # explicit index when indexing
#              --> # implicit index when slicing
# With loc
# --> # The 'loc' attribute allows indexing and slicing that always references the explicit index.
# With iloc
# --> # The 'iloc' attribute on the other hand allows indexing and slicing that always references implicit Python-styl
e index
```

Data Selection in DataFrame

In [ ]:
```
# DataFrame works like a two-dimensional or structured array, and in other ways like a dictionary of Series structures
--
# -- sharing the same index
```

In [29]:
```
import pandas as pd
```

In [ ]:

DataFrame as a dictionary:

In [30]:
```
states_capitals = {'Karnataka':'Bangalore', 'Andrapradesh':'Hyderabad', 'Tamilnadu':'Chennai', 'Keral':'Thiruvanathapu
ram',
                   'Maharastra':'Mumbai', 'India':'Dehli'}
data_capitals = pd.Series(states_capitals)
```

In [31]:
```
states_lang = {'Karnataka':'Kannada', 'Andrapradesh':'Telugu', 'Tamilnadu':'Tamil', 'Kerala':'Malayalam',
               'Maharastra':'hindi', 'Panjab':'Panjabi'}
sl_df = pd.Series(states_lang)
```

In [32]:
```
data = pd.DataFrame({'capitals': states_capitals, 'language': states_lang})
data
```

Out[32]:

	capitals	language
Andrapradesh	*Hyderabad*	*Telugu*
India	*Dehli*	*NaN*
Karnataka	*Bangalore*	*Kannada*
Keral	*Thiruvanathapuram*	*NaN*
Kerala	*NaN*	*Malayalam*
Maharastra	*Mumbai*	*hindi*
Panjab	*NaN*	*Panjabi*
Tamilnadu	*Chennai*	*Tamil*

In [33]:
```
# We can access the individual column data using dictionary style as

data['capitals']
```

Out[33]:
```
Andrapradesh            Hyderabad
India                       Dehli
Karnataka               Bangalore
Keral           Thiruvanathapuram
Kerala                        NaN
Maharastra                 Mumbai
Panjab                        NaN
Tamilnadu                  Chennai
Name: capitals, dtype: object
```

In [34]:
```
# Equivalently, we can use attribute-style access with column names that are strings:
# make sure that there should not be any space while creating the column names
# It is very important to remember that, we should always use identifier rules to create any string/dictionary names i
n Python

data.capitals
```

Out[34]:
```
Andrapradesh            Hyderabad
India                       Dehli
Karnataka               Bangalore
Keral           Thiruvanathapuram
Kerala                        NaN
Maharastra                 Mumbai
Panjab                        NaN
Tamilnadu                  Chennai
Name: capitals, dtype: object
```

In [35]:
```
# Both dictionary style access and the attribute style access remains same:

data['capitals'] is data.capitals
```

Out[35]:
```
True
```

In [ ]:
```
# The above methos doesn't work for all cases: Ex.: DataFrame has a pop method so, if the DataFrame contains pop
# column then accessing it via data.pop results false
```

In [ ]:
```
# data.pop is data['pop'] is not possible so in that case it is better to use data['pop'] instead of data.pop to acces
s data
# object.
```

In [ ]:
```
# Dictionary style syntax can also be used to modify the data object:
```

In [37]:
```
# Ex.:

s_sub = pd.Series({'Pruthvi': 'Kannada', 'Pranam': 'Hindi', 'Pratham':'English', 'Pravera':'Maths', 'Prabu':'Science'
})
total_m = pd.Series({'Pruthvi': 60, 'Pranam': 60, 'Pratham':60, 'Pravera':60, 'Prabu':60})
minf_m = pd.Series({'Pruthvi': 30, 'Pranam': 30, 'Pratham':30, 'Pravera':30, 'Prabu':30})
obt_m = pd.Series({'Pruthvi': 25, 'Pranam': 35, 'Pratham':40, 'Pravera':60, 'Prabu':55})
students_d = pd.DataFrame({'Sub' : s_sub, 'T_m' : total_m, 'Min_m' : minf_m, 'O_m' : obt_m})
students_d
```

Out[37]:

	Sub	T_m	Min_m	O_m
Pruthvi	*Kannada*	*60*	*30*	*25*
Pranam	*Hindi*	*60*	*30*	*35*
Pratham	*English*	*60*	*30*	*40*
Pravera	*Maths*	*60*	*30*	*60*
Prabu	*Science*	*60*	*30*	*55*

In [ ]:
```
# We can calculate the score obtained by the students using dictionary style access and create new object
# Like , students_d['score'] = students_d['obt_m'] / students_d['total_m']
# This means we can do element by element arithmitic operation between series objects using dictionary like syntax
```

In [38]:
```
# Create a new column name 'score'

students_d['score'] = students_d['O_m'] / students_d['T_m']
students_d
```

Out[38]:

	Sub	T_m	Min_m	O_m	score
Pruthvi	*Kannada*	*60*	*30*	*25*	*0.416667*
Pranam	*Hindi*	*60*	*30*	*35*	*0.583333*
Pratham	*English*	*60*	*30*	*40*	*0.666667*
Pravera	*Maths*	*60*	*30*	*60*	*1.000000*
Prabu	*Science*	*60*	*30*	*55*	*0.916667*

In [39]:
```
# Let's create a new column which has the minimum passing score

students_d['mp_score'] = students_d['Min_m'] / students_d['T_m']
students_d
```

Out[39]:

	Sub	T_m	Min_m	O_m	score	mp_score
Pruthvi	*Kannada*	*60*	*30*	*25*	*0.416667*	*0.5*
Pranam	*Hindi*	*60*	*30*	*35*	*0.583333*	*0.5*
Pratham	*English*	*60*	*30*	*40*	*0.666667*	*0.5*
Pravera	*Maths*	*60*	*30*	*60*	*1.000000*	*0.5*
Prabu	*Science*	*60*	*30*	*55*	*0.916667*	*0.5*

DataFrame as two-dimensional array:

In [40]:
```
# Let's see the DataFrame as an enhanced two-dimensional array: use 'values' attribute

students_d.values
```

Out[40]:
```
array([['Kannada', 60, 30, 25, 0.4166666666666667, 0.5],
       ['Hindi', 60, 30, 35, 0.5833333333333334, 0.5],
       ['English', 60, 30, 40, 0.6666666666666666, 0.5],
       ['Maths', 60, 30, 60, 1.0, 0.5],
       ['Science', 60, 30, 55, 0.9166666666666666, 0.5]], dtype=object)
```

In [41]:
```
# We can do many array like operations on DataFrame:

# Ex.:
# Let's transpose(swaping rows and columns) the entire DataFrame array values using 'T' attribute
# Let's see the signature of 'transpose'
pd.DataFrame.T?
```

In [42]:
```
students_d.T
```

Out[42]:

	Pruthvi	Pranam	Pratham	Pravera	Prabu
Sub	*Kannada*	*Hindi*	*English*	*Maths*	*Science*
T_m	*60*	*60*	*60*	*60*	*60*
Min_m	*30*	*30*	*30*	*30*	*30*
O_m	*25*	*35*	*40*	*60*	*55*
score	*0.416667*	*0.583333*	*0.666667*	*1*	*0.916667*
mp_score	*0.5*	*0.5*	*0.5*	*0.5*	*0.5*

In [43]:
```
#  We can pass a single index to an two-dimentional array to access the individual rows:

students_d.values[0]
```

Out[43]:
```
array(['Kannada', 60, 30, 25, 0.4166666666666667, 0.5], dtype=object)
```

In [44]:
```
students_d.values[1]
```

Out[44]:
```
array(['Hindi', 60, 30, 35, 0.5833333333333334, 0.5], dtype=object)
```

In [45]:
```
# To access the individual column we can pass a single 'index' to a DataFrame itself

students_d['score']
```

Out[45]:
```
Pruthvi    0.416667
Pranam     0.583333
Pratham    0.666667
Pravera    1.000000
Prabu      0.916667
Name: score, dtype: float64
```

In [46]:
```
students_d['T_m']
```

Out[46]:
```
Pruthvi    60
Pranam     60
Pratham    60
Pravera    60
Prabu      60
Name: T_m, dtype: int64
```

In [ ]:
```
# We can use loc, iloc and ix to access the objects of DataFrame similar to like series.
```

In [47]:
```
students_d
```

Out[47]:

	Sub	T_m	Min_m	O_m	score	mp_score
Pruthvi	*Kannada*	*60*	*30*	*25*	*0.416667*	*0.5*
Pranam	*Hindi*	*60*	*30*	*35*	*0.583333*	*0.5*
Pratham	*English*	*60*	*30*	*40*	*0.666667*	*0.5*
Pravera	*Maths*	*60*	*30*	*60*	*1.000000*	*0.5*
Prabu	*Science*	*60*	*30*	*55*	*0.916667*	*0.5*

In [48]:
```
# First, while indexing refers to columns, slicing refers to rows:

# To access the rows simply we can pass the range index to a DataFrame : This is slicing
# This implicit index slicing doesn't include last index
students_d[1:3]
```

Out[48]:

	Sub	T_m	Min_m	O_m	score	mp_score
Pranam	*Hindi*	*60*	*30*	*35*	*0.583333*	*0.5*
Pratham	*English*	*60*	*30*	*40*	*0.666667*	*0.5*

In [ ]:
```
# students_d['T_m':'score'] # if you run this cell this throws Key Error because we are trying to slice the columns
#students_d['T_m':'score']
```

In [49]:
```
# The explicit slicing does innlude the last index

students_d['Pranam':'Pravera']
```

Out[49]:

	Sub	T_m	Min_m	O_m	score	mp_score
Pranam	*Hindi*	*60*	*30*	*35*	*0.583333*	*0.5*
Pratham	*English*	*60*	*30*	*40*	*0.666667*	*0.5*
Pravera	*Maths*	*60*	*30*	*60*	*1.000000*	*0.5*

In [ ]:
```
# "iloc" helps to slice the DataFrame objects implicitly as like sereis
# "iloc" for position-based indexing
```

In [ ]:
```
# Let's see the signature of 'loc', 'iloc' and 'ix'
```

In [50]:
```
pd.DataFrame.loc?
```

In [51]:
```
pd.DataFrame.iloc?
```

In [52]:
```
pd.DataFrame.ix?
```

In [53]:
```
print(students_d)
# Let's slice the frist 3 rows and frist 2 columns
students_d.iloc[0:3, 0:2]
```

```
             Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   25  0.416667       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
```

Out[53]:

	Sub	T_m
Pruthvi	*Kannada*	*60*
Pranam	*Hindi*	*60*
Pratham	*English*	*60*

In [54]:
```
# Let's slice from 2nd row to 5th row and 2nd column to end of the column
print(students_d)
students_d.iloc[2:5, 2:]
```

```
             Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   25  0.416667       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
```

Out[54]:

	Min_m	O_m	score	mp_score
Pratham	*30*	*40*	*0.666667*	*0.5*
Pravera	*30*	*60*	*1.000000*	*0.5*
Prabu	*30*	*55*	*0.916667*	*0.5*

In [ ]:
```
# We can use "loc" to explicitly loc the rows and columns
# "loc" for label based indexing
```

In [55]:
```
print(students_d)
students_d.loc[:, :]
```

```
             Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   25  0.416667       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
```

Out[55]:

	Sub	T_m	Min_m	O_m	score	mp_score
Pruthvi	*Kannada*	*60*	*30*	*25*	*0.416667*	*0.5*
Pranam	*Hindi*	*60*	*30*	*35*	*0.583333*	*0.5*
Pratham	*English*	*60*	*30*	*40*	*0.666667*	*0.5*
Pravera	*Maths*	*60*	*30*	*60*	*1.000000*	*0.5*
Prabu	*Science*	*60*	*30*	*55*	*0.916667*	*0.5*

In [56]:
```
# Explicit slice does inlude the last index value
print(students_d)
students_d.loc[:'Pravera', 'O_m':]
```

```
             Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   25  0.416667       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
```

Out[56]:

	O_m	score	mp_score
Pruthvi	*25*	*0.416667*	*0.5*
Pranam	*35*	*0.583333*	*0.5*
Pratham	*40*	*0.666667*	*0.5*
Pravera	*60*	*1.000000*	*0.5*

In [57]:
```
# Explicit slice does inlude the last index value
print(students_d)
students_d.loc['Pranam':'Pravera', 'T_m':'score']
```

```
             Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   25  0.416667       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
```

Out[57]:

	T_m	Min_m	O_m	score
Pranam	*60*	*30*	*35*	*0.583333*
Pratham	*60*	*30*	*40*	*0.666667*
Pravera	*60*	*30*	*60*	*1.000000*

In [58]:
```
# The "ix" indexer allows a hybrid of "loc" and "iloc" features
# But "ix" is depricated only loc and iloc are in use

students_d.ix[:3, 'T_m':'score']
```

```
C:\Users\user\Anaconda3\lib\site-packages\ipykernel_launcher.py:4: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  after removing the cwd from sys.path.
```

Out[58]:

	T_m	Min_m	O_m	score
Pruthvi	*60*	*30*	*25*	*0.416667*
Pranam	*60*	*30*	*35*	*0.583333*
Pratham	*60*	*30*	*40*	*0.666667*

In [59]:
```
# We can do masking and fancy indexing using "loc" attribute for labeled index

students_d.loc[students_d['O_m'] > 30, ['Min_m', 'O_m']]
```

Out[59]:

	Min_m	O_m
Pranam	*30*	*35*
Pratham	*30*	*40*
Pravera	*30*	*60*
Prabu	*30*	*55*

In [60]:
```
# We can do modification using "iloc" attribute for position based index
print(students_d)
students_d.iloc[0, 3] = 30

students_d['score'] = students_d['O_m'] / students_d['T_m']
print(students_d)
```

```
            Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   25  0.416667       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
            Sub  T_m  Min_m  O_m     score  mp_score
Pruthvi  Kannada   60     30   30  0.500000       0.5
Pranam     Hindi   60     30   35  0.583333       0.5
Pratham  English   60     30   40  0.666667       0.5
Pravera    Maths   60     30   60  1.000000       0.5
Prabu    Science   60     30   55  0.916667       0.5
```

In [ ]:

In [ ]:

In [ ]: