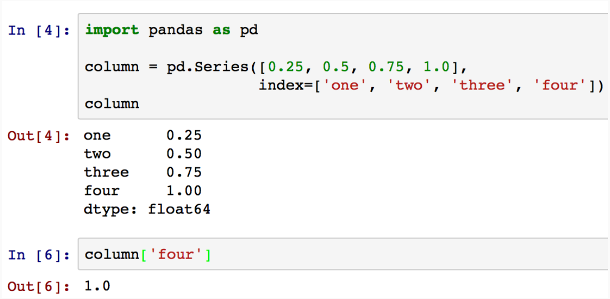
INTRODUCTION TO pandas

pandas objects can be thought of as “enhanced” versions of numpy arrays in which the rows and columns are identified with labels rather than integers. Basics pandas objects:

- **Series:** 1D array of data with an index object (labels).



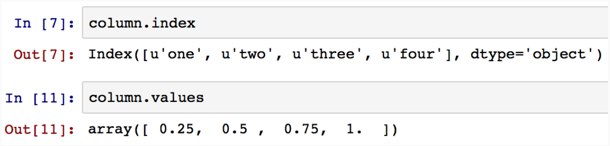
```
In [4]:  import pandas as pd

         column = pd.Series([0.25, 0.5, 0.75, 1.0],
                            index=['one', 'two', 'three', 'four'])
         column

Out[4]:  one      0.25
         two      0.50
         three    0.75
         four     1.00
         dtype: float64

In [6]:  column['four']

Out[6]:  1.0
```

Each series has a “values” component and an “index” component. Series come with the same built-in functions, like mean, min, std, as numpy arrays.

INTRODUCTION TO pandas

pandas objects can be thought of as "enhanced" versions of numpy arrays in which the rows and columns are identified with labels rather than integers. Basics pandas objects:

- **Series:** 1D array of data with an index object (labels).



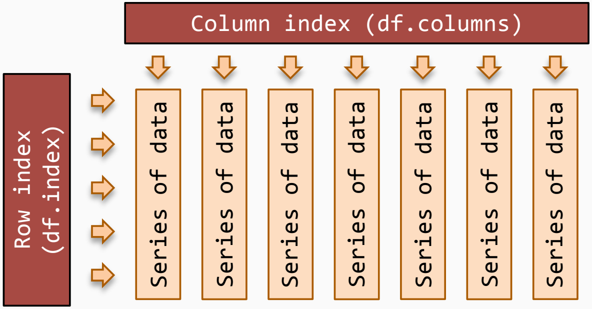
```
In [7]:  column.index
Out[7]:  Index([u'one', u'two', u'three', u'four'], dtype='object')

In [11]: column.values
Out[11]: array([ 0.25,  0.5 ,  0.75,  1.  ])
```

Each series has a "values" component and an "index" component. Series come with the same built-in functions, like mean, min, std, as numpy arrays.

INTRODUCTION TO pandas

pandas objects can be thought of as “enhanced” versions of **numpy** arrays in which the rows and columns are identified with labels rather than integers. Basics **pandas** objects:

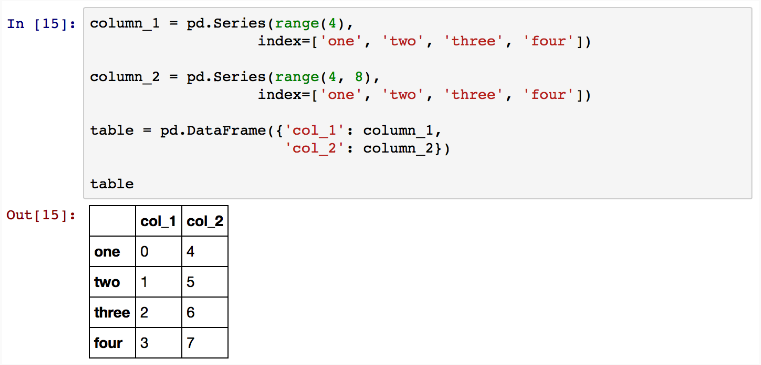
- **DataFrame:** 2D table of data with column and row index objects (labels).



Each column in the data frame is a *series*.

GETTING STARTED WITH pandas

We can create a data frame from columns (series objects):



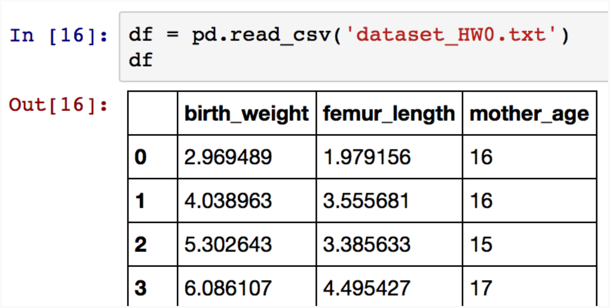
```
In [15]: column_1 = pd.Series(range(4),
                              index=['one', 'two', 'three', 'four'])

         column_2 = pd.Series(range(4, 8),
                              index=['one', 'two', 'three', 'four'])

         table = pd.DataFrame({'col_1': column_1,
                                'col_2': column_2})

         table
```

Out[15]:

	col_1	col_2
one	0	4
two	1	5
three	2	6
four	3	7

GETTING STARTED WITH pandas

We can import tabular data in a csv file into a data frame:



```
In [16]: df = pd.read_csv('dataset_HW0.txt')
         df
```

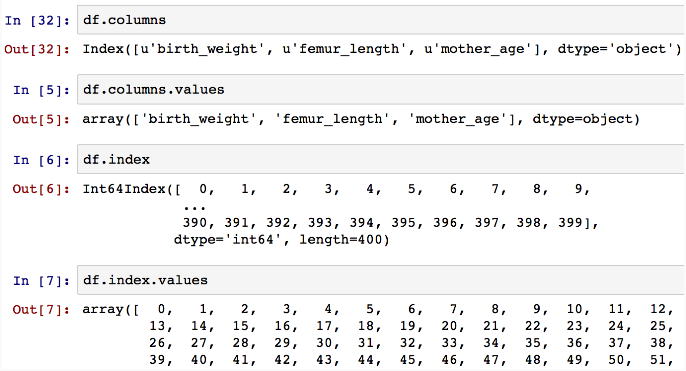
Out[16]:

	birth_weight	femur_length	mother_age
0	2.969489	1.979156	16
1	4.038963	3.555681	16
2	5.302643	3.385633	15
3	6.086107	4.495427	17

GETTING TO KNOW YOUR DATAFRAME

We should start by getting a rough sense of what's in the data

- **The indices of your data frame:**



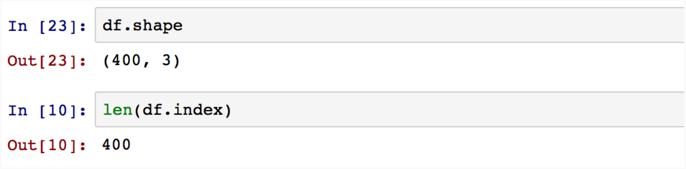
```
In [32]: df.columns
Out[32]: Index([u'birth_weight', u'femur_length', u'mother_age'], dtype='object')

 In [5]: df.columns.values
 Out[5]: array(['birth_weight', 'femur_length', 'mother_age'], dtype=object)

 In [6]: df.index
 Out[6]: Int64Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
                    ...
                    390, 391, 392, 393, 394, 395, 396, 397, 398, 399],
                   dtype='int64', length=400)

 In [7]: df.index.values
 Out[7]: array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
                 13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
                 26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
                 39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
```

The `.index` and `.columns` attributes give access to the index objects of rows and columns (resp).

GETTING TO KNOW YOUR DATAFRAME

We should start by getting a rough sense of what's in the data

- **The shape of your data frame:**

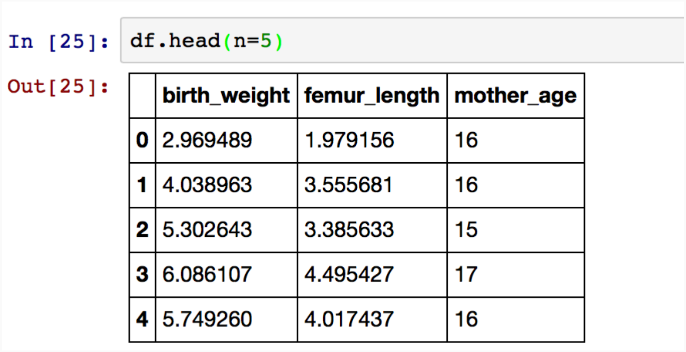


```
In [23]: df.shape

Out[23]: (400, 3)

In [10]: len(df.index)

Out[10]: 400
```

GETTING TO KNOW YOUR DATAFRAME

We should start by getting a rough sense of what's in the data

- **The first entries in your data frame:**



```
In [25]: df.head(n=5)
```

Out[25]:

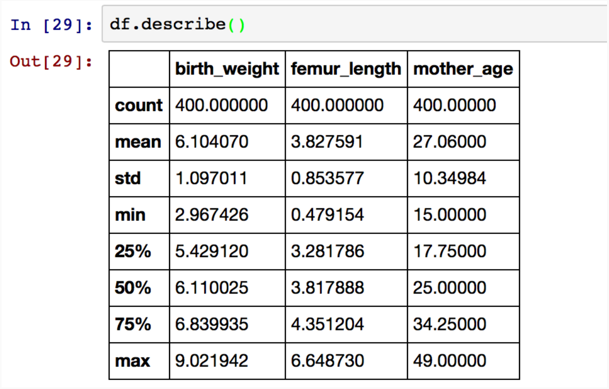
	birth_weight	femur_length	mother_age
0	2.969489	1.979156	16
1	4.038963	3.555681	16
2	5.302643	3.385633	15
3	6.086107	4.495427	17
4	5.749260	4.017437	16

The `.head()` function returns a (row-wise) truncated version of your data frame!

GETTING TO KNOW YOUR DATAFRAME

We should start by getting a rough sense of what's in the data

- A summary of your data frame:



```
In [29]: df.describe()
```

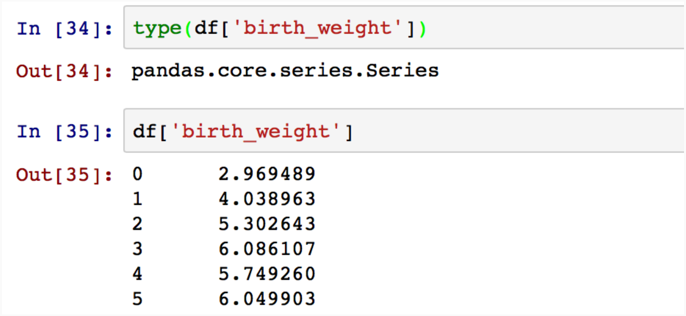
Out[29]:

	birth_weight	femur_length	mother_age
count	400.000000	400.000000	400.00000
mean	6.104070	3.827591	27.06000
std	1.097011	0.853577	10.34984
min	2.967426	0.479154	15.00000
25%	5.429120	3.281786	17.75000
50%	6.110025	3.817888	25.00000
75%	6.839935	4.351204	34.25000
max	9.021942	6.648730	49.00000

The `.describe()` function returns all the descriptive stats for each column as a data frame object!

ACCESSING COLUMNS & ROWS

- Accessing a column by label:

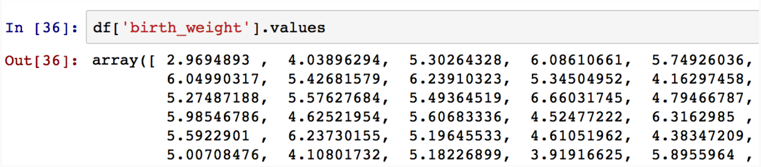


```
In [34]: type(df['birth_weight'])

Out[34]: pandas.core.series.Series

In [35]: df['birth_weight']

Out[35]: 0       2.969489
         1       4.038963
         2       5.302643
         3       6.086107
         4       5.749260
         5       6.049903
```

You can access a column by it's column name or position (you can also access a *list* of columns)!

ACCESSING COLUMNS & ROWS

- Accessing the values of column:

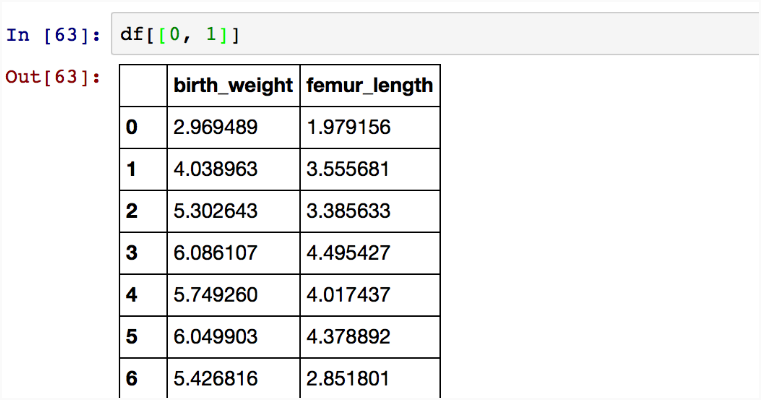


```
In [36]: df['birth_weight'].values

Out[36]: array([ 2.9694893 ,  4.03896294,  5.30264328,  6.08610661,  5.74926036,
                 6.04990317,  5.42681579,  6.23910323,  5.34504952,  4.16297458,
                 5.27487188,  5.57627684,  5.49364519,  6.66031745,  4.79466787,
                 5.98546786,  4.62521954,  5.60683336,  4.52477222,  6.3162985 ,
                 5.5922901 ,  6.23730155,  5.19645533,  4.61051962,  4.38347209,
                 5.00708476,  4.10801732,  5.18226899,  3.91916625,  5.8955964 ,
```

You can access a column by it's column name or position (you can also access a *list* of columns)!

ACCESSING COLUMNS & ROWS

- Accessing columns by position:



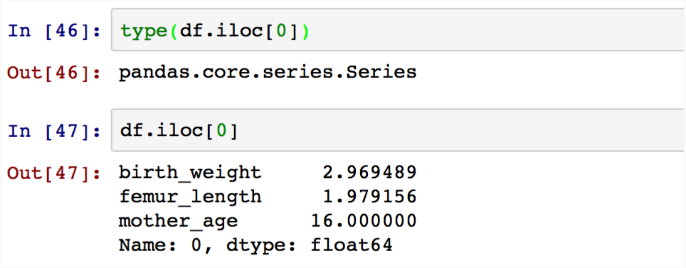
```
In [63]: df[[0, 1]]
Out[63]:
```

	birth_weight	femur_length
0	2.969489	1.979156
1	4.038963	3.555681
2	5.302643	3.385633
3	6.086107	4.495427
4	5.749260	4.017437
5	6.049903	4.378892
6	5.426816	2.851801

You can access a column by it's column name or position (you can also access a *list* of columns)!

ACCESSING COLUMNS & ROWS

- Accessing a row by position:

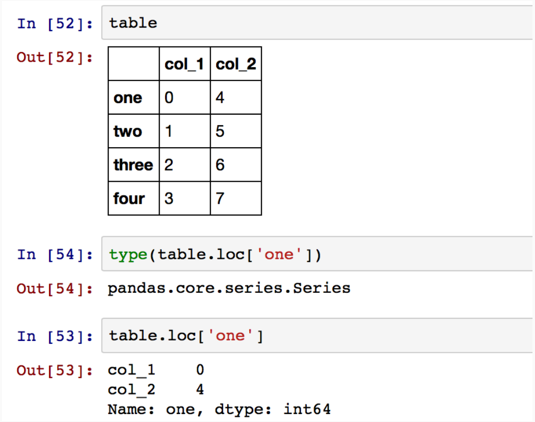


```
In [46]: type(df.iloc[0])

Out[46]: pandas.core.series.Series

In [47]: df.iloc[0]

Out[47]: birth_weight     2.969489
         femur_length     1.979156
         mother_age      16.000000
         Name: 0, dtype: float64
```

You can access a column by it’s row name or position!

ACCESSING COLUMNS & ROWS

- Accessing a row by label:



```
In [52]: table
Out[52]:
```

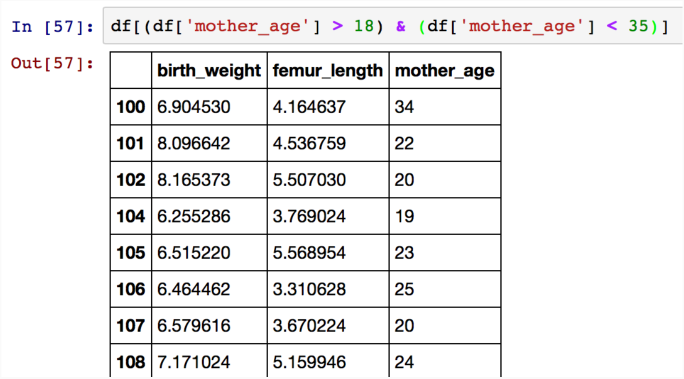
	col_1	col_2
one	0	4
two	1	5
three	2	6
four	3	7

```
In [54]: type(table.loc['one'])
Out[54]: pandas.core.series.Series

In [53]: table.loc['one']
Out[53]: col_1     0
         col_2     4
         Name: one, dtype: int64
```

You can access a column by it's row name or position!

FILTERING

Filtering works very much like with `numpy` arrays!



```
In [57]: df[(df['mother_age'] > 18) & (df['mother_age'] < 35)]
```

Out[57]:

	birth_weight	femur_length	mother_age
100	6.904530	4.164637	34
101	8.096642	4.536759	22
102	8.165373	5.507030	20
104	6.255286	3.769024	19
105	6.515220	5.568954	23
106	6.464462	3.310628	25
107	6.579616	3.670224	20
108	7.171024	5.159946	24