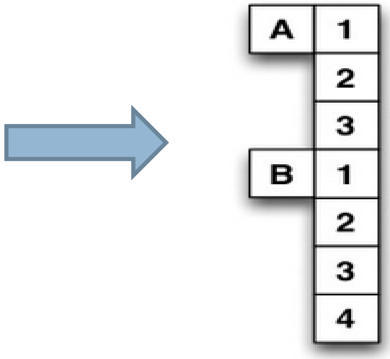
MINERÍA DE DATOS DESCRIPTIVA LIBRERIA PANDAS DE PYTHON

Pandas: Panel Data System

- Python library to provide data analysis features
- Built on top of NumPy (SciPy, matplotlib)
- Key components provided by pandas:
  - Series
  - DataFrame
- The ideal tool for data scientists
  - Munging data
  - Cleaning data
  - Analyzing
  - Modeling
  - Organizing the result of the analysis into a suitable form for plotting or tabular display

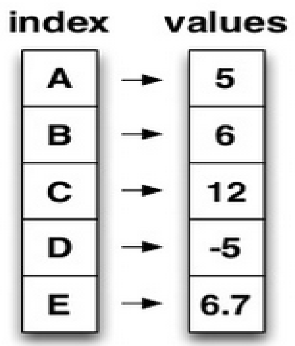
Pandas: Axis Indexing

- Every axis has an index
- Highly optimized structure
- Hierarchical indexing
  - Semantics: a tuple at each tick
  - Enables easy group by selection
- Group by and join-type operations



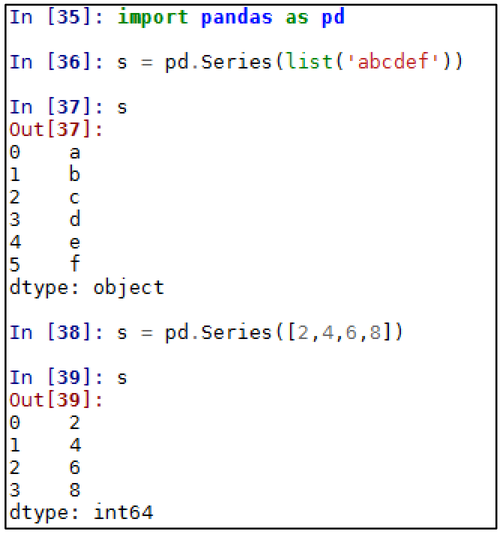
Pandas: Series

- One-dimensional array-like object containing data and labels
  - Sublass of numpy.ndarray
  - Data: can be of any type
  - Index labels need not be ordered
  - Duplicates are possible at the cost of a reduced functionality

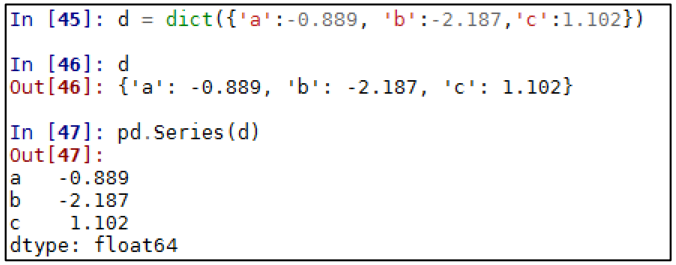


Pandas: Series construction

- Lot of ways to build Series
  - From lists
  - From dictionaries



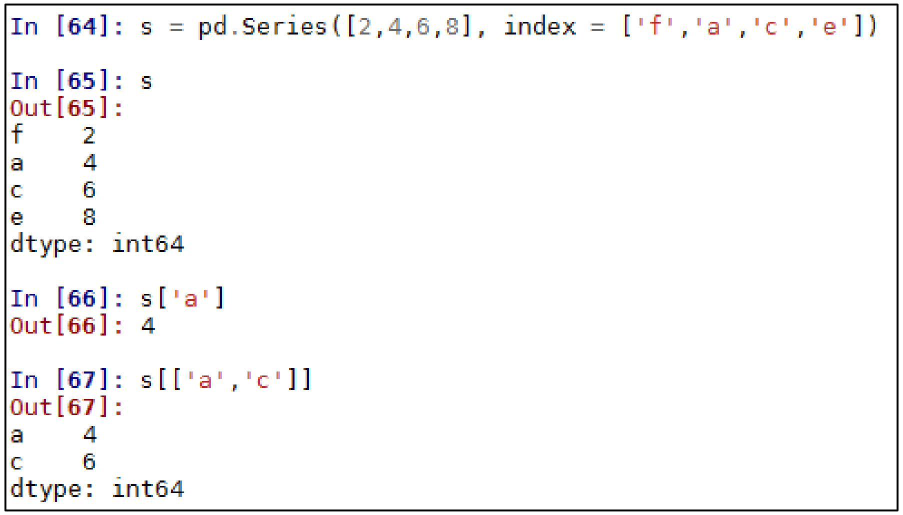
```
In [35]: import pandas as pd

In [36]: s = pd.Series(list('abcdef'))

In [37]: s
Out[37]:
0     a
1     b
2     c
3     d
4     e
5     f
dtype: object

In [38]: s = pd.Series([2,4,6,8])

In [39]: s
Out[39]:
0     2
1     4
2     6
3     8
dtype: int64
```



```
In [45]: d = dict({'a':-0.889, 'b':-2.187,'c':1.102})

In [46]: d
Out[46]: {'a': -0.889, 'b': -2.187, 'c': 1.102}

In [47]: pd.Series(d)
Out[47]:
a   -0.889
b   -2.187
c    1.102
dtype: float64
```

Pandas: Series indexing

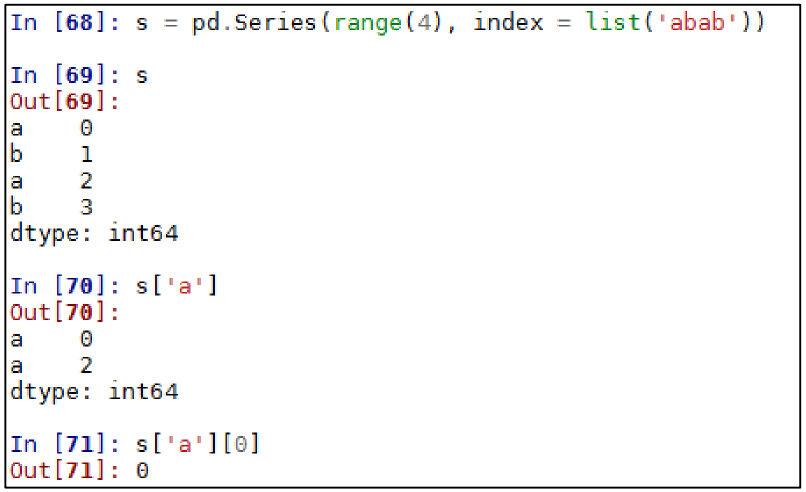
- A Series index can be specified
  - Single values can be selected by index
  - Multiple values can be selected with multiple indexes



```
In [64]: s = pd.Series([2,4,6,8], index = ['f','a','c','e'])

In [65]: s
Out[65]:
f    2
a    4
c    6
e    8
dtype: int64

In [66]: s['a']
Out[66]: 4

In [67]: s[['a','c']]
Out[67]:
a    4
c    6
dtype: int64
```

Pandas: Series indexing

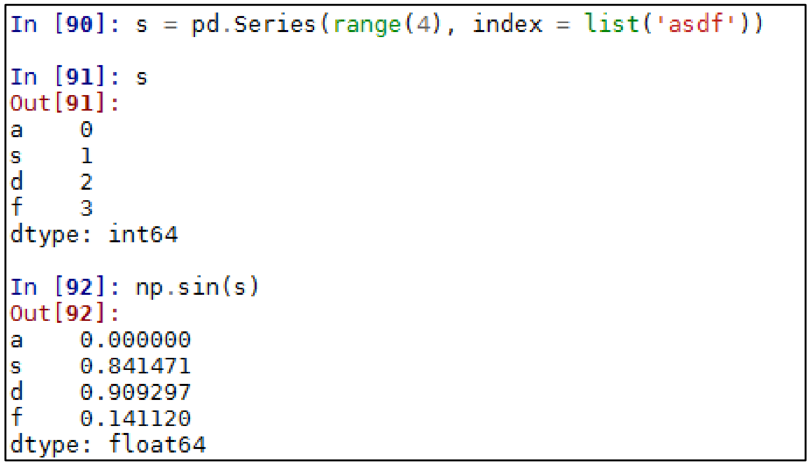
- Think of Series as a fixed-length, ordered dictionary
  - However index items do not have to be unique



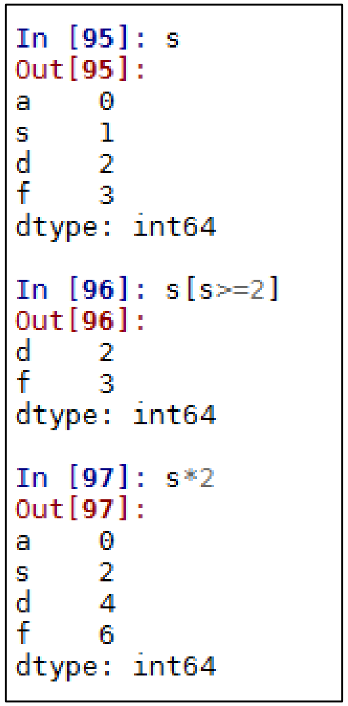
```
In [68]: s = pd.Series(range(4), index = list('abab'))

In [69]: s
Out[69]:
a    0
b    1
a    2
b    3
dtype: int64

In [70]: s['a']
Out[70]:
a    0
a    2
dtype: int64

In [71]: s['a'][0]
Out[71]: 0
```

Pandas: Series operations

- Series works with NumPy
  - Filtering
  - Mathematical operations



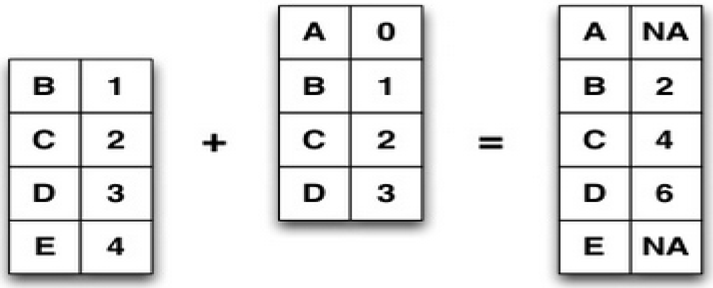
```
In [90]: s = pd.Series(range(4), index = list('asdf'))

In [91]: s
Out[91]:
a    0
s    1
d    2
f    3
dtype: int64

In [92]: np.sin(s)
Out[92]:
a    0.000000
s    0.841471
d    0.909297
f    0.141120
dtype: float64
```

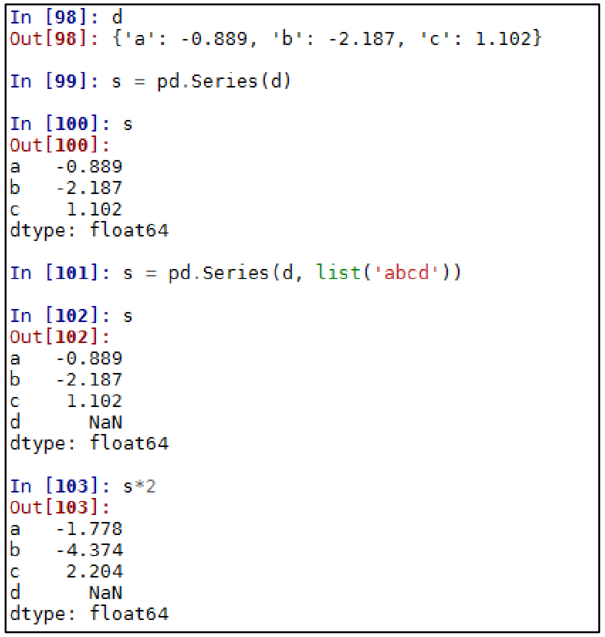


```
In [95]: s
Out[95]:
a    0
s    1
d    2
f    3
dtype: int64

In [96]: s[s>=2]
Out[96]:
d    2
f    3
dtype: int64

In [97]: s*2
Out[97]:
a    0
s    2
d    4
f    6
dtype: int64
```

Pandas: Series (missing data)

- Series can accommodate incomplete data
- Unlike in a NumPy ndarray, data is automatically aligned
  - Binary operations are joins

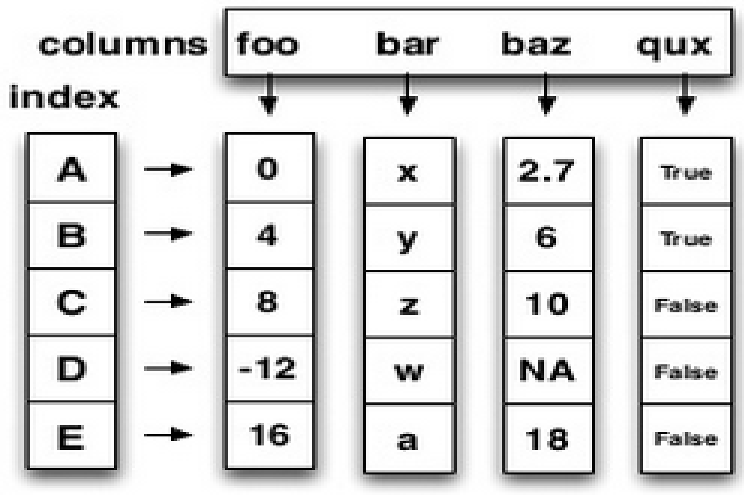




```
In [98]: d
Out[98]: {'a': -0.889, 'b': -2.187, 'c': 1.102}

In [99]: s = pd.Series(d)

In [100]: s
Out[100]:
a   -0.889
b   -2.187
c    1.102
dtype: float64

In [101]: s = pd.Series(d, list('abcd'))

In [102]: s
Out[102]:
a   -0.889
b   -2.187
c    1.102
d      NaN
dtype: float64

In [103]: s*2
Out[103]:
a   -1.778
b   -4.374
c    2.204
d      NaN
dtype: float64
```

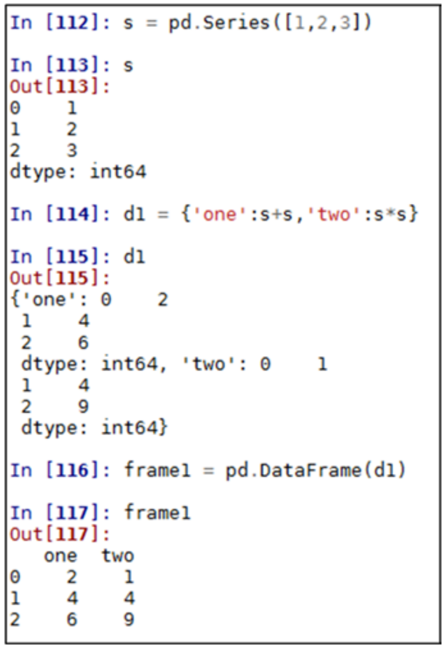
Pandas: DataFrames

- Spreadsheet-like data structure containing an ordered collection of columns
  - Data that can be represented as tables: Rows and columns (with their own indexes)
    - Each row is a different object
    - Columns represent attributes of the objects
  - Each column can have a different type
    - But a column is homogenously typed
- Consider as a dictionary of Series (with shared index)
- Size mutable: insert and delete columns

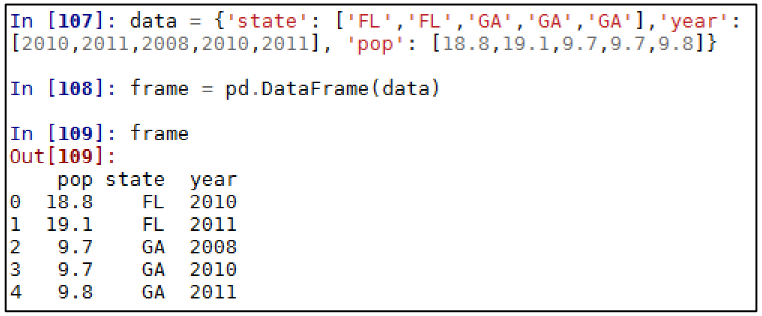


Pandas: DataFrames generation

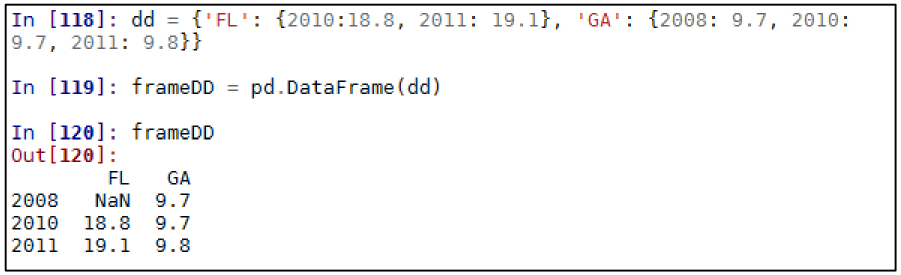
- DataFrame generation from a dictionary of equal-length lists
- Generation from a dictionary of dictionaries



```
In [112]: s = pd.Series([1,2,3])

In [113]: s
Out[113]:
0    1
1    2
2    3
dtype: int64

In [114]: d1 = {'one':s+s,'two':s*s}

In [115]: d1
Out[115]:
{'one': 0    2
 1    4
 2    6
 dtype: int64, 'two': 0    1
 1    4
 2    9
 dtype: int64}

In [116]: frame1 = pd.DataFrame(d1)

In [117]: frame1
Out[117]:
   one  two
0    2    1
1    4    4
2    6    9
```



```
In [107]: data = {'state': ['FL','FL','GA','GA','GA'],'year':
[2010,2011,2008,2010,2011], 'pop': [18.8,19.1,9.7,9.7,9.8]}

In [108]: frame = pd.DataFrame(data)

In [109]: frame
Out[109]:
    pop state  year
0  18.8    FL  2010
1  19.1    FL  2011
2   9.7    GA  2008
3   9.7    GA  2010
4   9.8    GA  2011
```



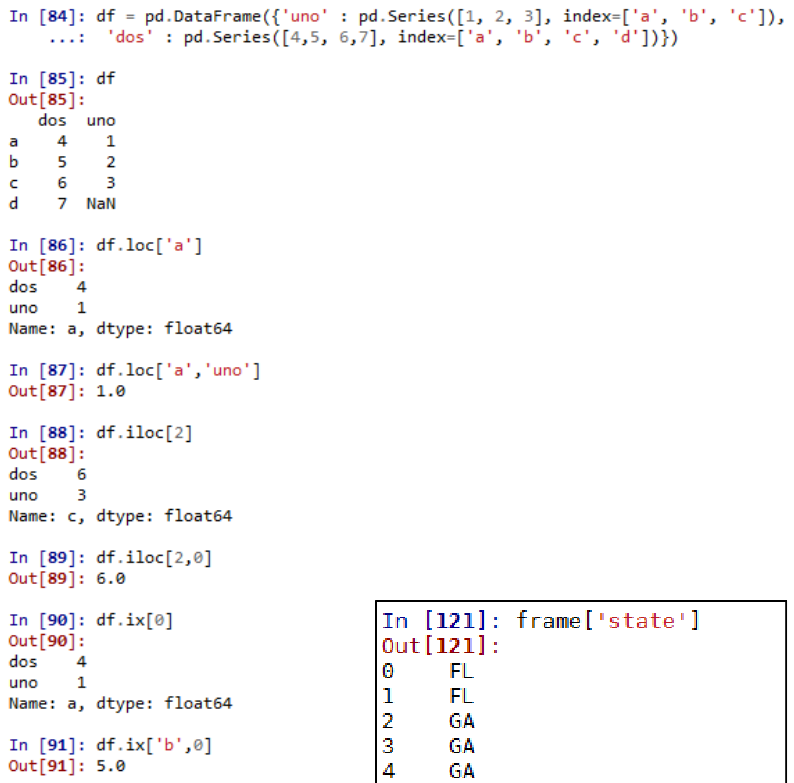
```
In [118]: dd = {'FL': {2010:18.8, 2011: 19.1}, 'GA': {2008: 9.7, 2010:
9.7, 2011: 9.8}}

In [119]: frameDD = pd.DataFrame(dd)

In [120]: frameDD
Out[120]:
        FL   GA
2008   NaN  9.7
2010  18.8  9.7
2011  19.1  9.8
```

Pandas: Data loading

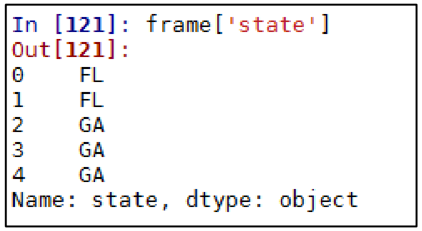
- Pandas supports several ways to handle data loading
- Text file data
  - read_csv
    - filepath_or_buffer: the file to read (it can be a URL)
    - sep : Delimiter to use (default ‘,’)
    - header: number of row where the column names are specified (default 0)
      - If there are no column names set it to None
    - names: list of the column names (header set to 0 to change them)
    - nrows : Number of rows of file to read. Useful for reading pieces of large files (default None: all file)
    - More information at this website
  - to_csv
    - filepath: name of the file(‘fileName.csv')
      - df.to_csv(‘example.csv’): writes the content of the DataFrame df in the file example.csv
  - Structured data: JSON, XML, HTML
  - Specific libraries
- Excel and database

Pandas: DataFrames slicing

- Select column by name
- To select data we can use the methods:
  - *loc* allows one to select data using row and column labels
  - *iloc* allows one to select data using row and column position
  - *ix* uses both labels and positions
- Using DataFrames they can return
  - A Series if only the row position is specified
  - A single value if the row and column are spefified
    - In the case of a Series they return a single value



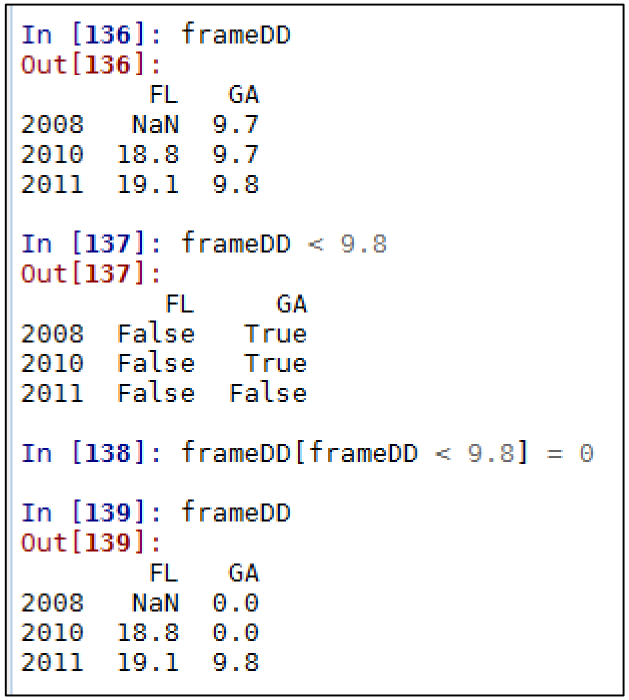
```
In [84]: df = pd.DataFrame({'uno' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
    ...:  'dos' : pd.Series([4,5, 6,7], index=['a', 'b', 'c', 'd'])})

In [85]: df
Out[85]:
   dos  uno
a    4    1
b    5    2
c    6    3
d    7  NaN

In [86]: df.loc['a']
Out[86]:
dos    4
uno    1
Name: a, dtype: float64

In [87]: df.loc['a','uno']
Out[87]: 1.0

In [88]: df.iloc[2]
Out[88]:
dos    6
uno    3
Name: c, dtype: float64

In [89]: df.iloc[2,0]
Out[89]: 6.0

In [90]: df.ix[0]
Out[90]:
dos    4
uno    1
Name: a, dtype: float64

In [91]: df.ix['b',0]
Out[91]: 5.0
```



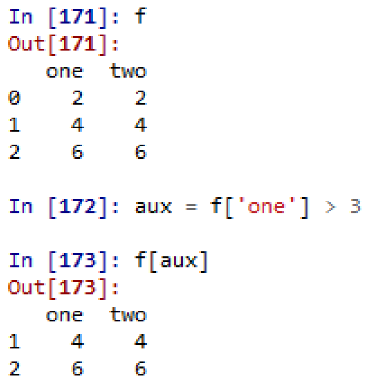
```
In [121]: frame['state']
Out[121]:
0     FL
1     FL
2     GA
3     GA
4     GA
Name: state, dtype: object
```

Pandas: DataFrames indexing

- Boolean indexing



```
In [136]: frameDD
Out[136]:
        FL   GA
2008   NaN  9.7
2010  18.8  9.7
2011  19.1  9.8

In [137]: frameDD < 9.8
Out[137]:
         FL     GA
2008  False   True
2010  False   True
2011  False  False

In [138]: frameDD[frameDD < 9.8] = 0

In [139]: frameDD
Out[139]:
        FL   GA
2008   NaN  0.0
2010  18.8  0.0
2011  19.1  9.8
```



```
In [171]: f
Out[171]:
   one  two
0    2    2
1    4    4
2    6    6

In [172]: aux = f['one'] > 3

In [173]: f[aux]
Out[173]:
   one  two
1    4    4
2    6    6
```

Pandas: DataFrames indexing

- You can obtain all the indexes using the index attribute
- The index can be changed: set_index function
  - set_index returns a new DataFrame
  - If you want to modify the actual one you have to specify it setting the inplace parameter to True
- The index can be reset to the original one: reset_index

```
print(users.set_index('user_id').head())
print('\n')

print(users.head())
print("\n^^^ I didn't actually change the DataFrame. ^^^\n")
```

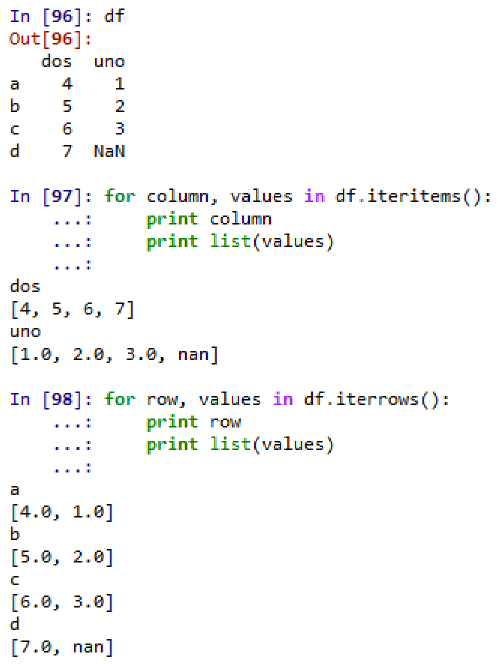
```
         age sex  occupation zip_code
user_id
1         24   M  technician    85711
2         53   F       other    94043
3         23   M      writer    32067
4         24   M  technician    43537
5         33   F       other    15213


   user_id  age sex  occupation zip_code
0        1   24   M  technician    85711
1        2   53   F       other    94043
2        3   23   M      writer    32067
3        4   24   M  technician    43537
4        5   33   F       other    15213
```

```
users.set_index('user_id', inplace=True)
users.head()
```

	age	sex	occupation	zip_code
user_id				
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067
4	24	M	technician	43537
5	33	F	other	15213

Pandas: DataFrames iterating

- *iteritems* and *iterrows* allow one to iterate over a DataFrame
  - Iteritems returns the DataFrame by column
  - Iterrows returns the DataFrame by rows



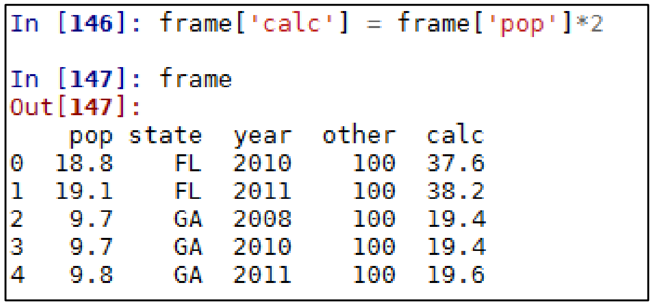
```
In [96]: df
Out[96]:
   dos  uno
a    4    1
b    5    2
c    6    3
d    7  NaN

In [97]: for column, values in df.iteritems():
    ...:     print column
    ...:     print list(values)
    ...:
dos
[4, 5, 6, 7]
uno
[1.0, 2.0, 3.0, nan]

In [98]: for row, values in df.iterrows():
    ...:     print row
    ...:     print list(values)
    ...:
a
[4.0, 1.0]
b
[5.0, 2.0]
c
[6.0, 3.0]
d
[7.0, nan]
```

Pandas: DataFrames visualization

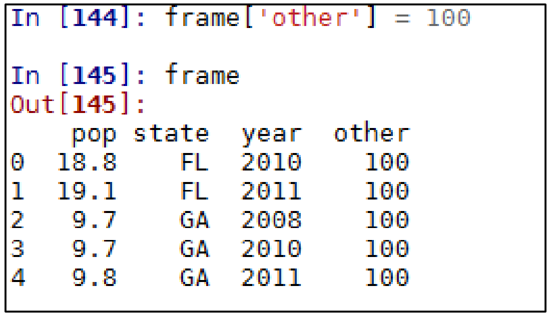
- The function head() displays the first five records of the dataset
  - The function tail() displays the last five
  - A different number of records (n) can be specified by parameter
    - head(n) or tail(n)
- In both cases you can visualize the data of only one field or a set of fields
  - df[[‘field’]].head() or df[[‘field’,’field1’]].head()
- You can also make a selection of the fields using conditions
  - df[df.‘field’ > X].head()

Pandas: DataFrames resizing

- Column addition
  - By computation
  - By direct assignment



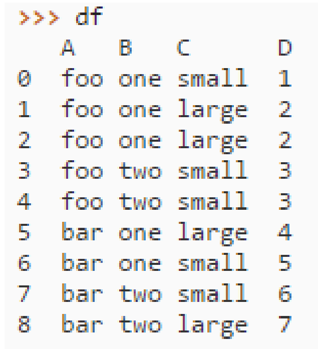
```
In [146]: frame['calc'] = frame['pop']*2

In [147]: frame
Out[147]:
    pop state  year  other  calc
0  18.8    FL  2010    100  37.6
1  19.1    FL  2011    100  38.2
2   9.7    GA  2008    100  19.4
3   9.7    GA  2010    100  19.4
4   9.8    GA  2011    100  19.6
```



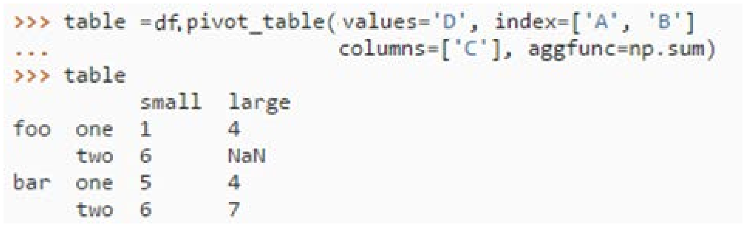
```
In [144]: frame['other'] = 100

In [145]: frame
Out[145]:
    pop state  year  other
0  18.8    FL  2010    100
1  19.1    FL  2011    100
2   9.7    GA  2008    100
3   9.7    GA  2010    100
4   9.8    GA  2011    100
```

Pandas: DataFrames reshaping

- Pivot tables: you can create a table having the result of an aggregation over specific fields
  - data.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')
    - Data: a DataFrame
    - Values: the column used to perform the aggregation
    - Index: the field(s) used as index in the pivot table
    - Columns: the field(s) to use as columns in the pivot table
      - Index and columns will be used to group the values and perform the aggregation over them
    - Aggfunc: the aggregation to apply
    - Fill_value: the value used to field empty fields



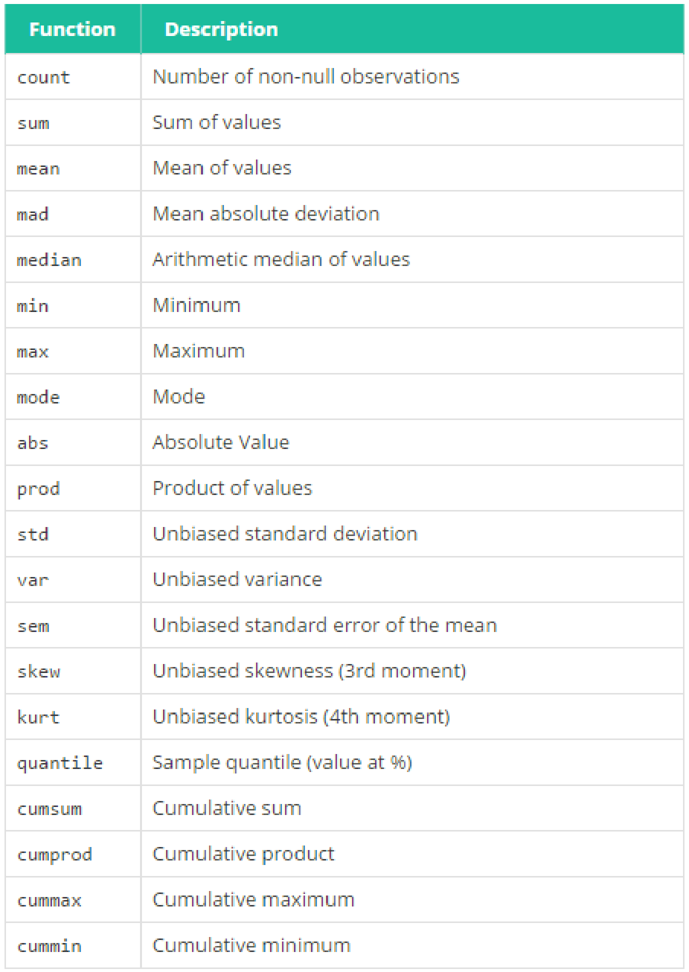
```
>>> df
     A    B    C      D
0   foo  one  small  1
1   foo  one  large  2
2   foo  one  large  2
3   foo  two  small  3
4   foo  two  small  3
5   bar  one  large  4
6   bar  one  small  5
7   bar  two  small  6
8   bar  two  large  7
```



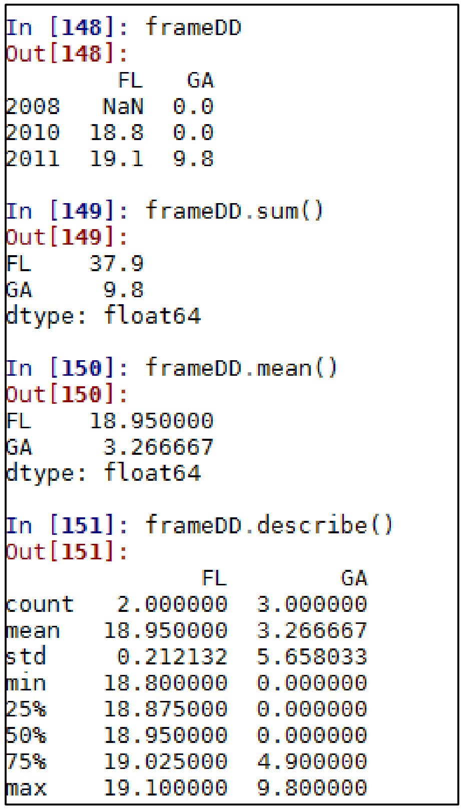
```
>>> table =df.pivot_table(values='D', index=['A', 'B']
...                       columns=['C'], aggfunc=np.sum)
>>> table
           small  large
foo  one   1      4
     two   6      NaN
bar  one   5      4
     two   6      7
```

Pandas: DataFrames operations

- Descriptive statistics



Function	Description
`count`	Number of non-null observations
`sum`	Sum of values
`mean`	Mean of values
`mad`	Mean absolute deviation
`median`	Arithmetic median of values
`min`	Minimum
`max`	Maximum
`mode`	Mode
`abs`	Absolute Value
`prod`	Product of values
`std`	Unbiased standard deviation
`var`	Unbiased variance
`sem`	Unbiased standard error of the mean
`skew`	Unbiased skewness (3rd moment)
`kurt`	Unbiased kurtosis (4th moment)
`quantile`	Sample quantile (value at %)
`cumsum`	Cumulative sum
`cumprod`	Cumulative product
`cummax`	Cumulative maximum
`cummin`	Cumulative minimum



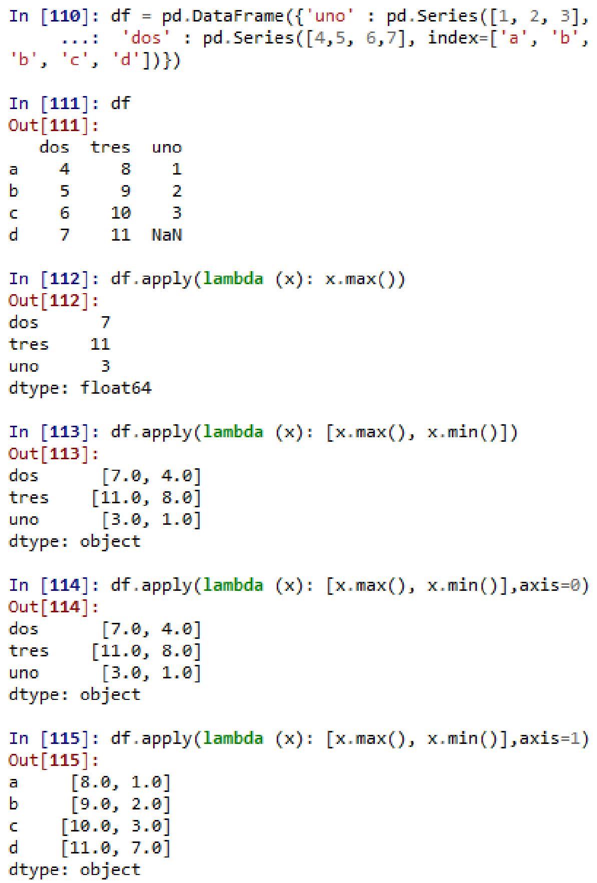
```
In [148]: frameDD
Out[148]:
        FL   GA
2008   NaN  0.0
2010  18.8  0.0
2011  19.1  9.8

In [149]: frameDD.sum()
Out[149]:
FL    37.9
GA     9.8
dtype: float64

In [150]: frameDD.mean()
Out[150]:
FL    18.950000
GA     3.266667
dtype: float64

In [151]: frameDD.describe()
Out[151]:
              FL         GA
count   2.000000   3.000000
mean   18.950000   3.266667
std     0.212132   5.658033
min    18.800000   0.000000
25%    18.875000   0.000000
50%    18.950000   0.000000
75%    19.025000   4.900000
max    19.100000   9.800000
```

Pandas: DataFrames operations

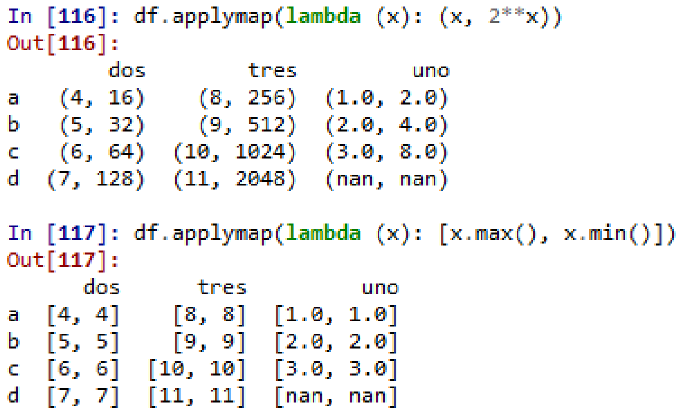
- More functions
  - Size: it counts the total number of values (not by columns)
  - Sort_values: to sort a Series
    - If you have a DataFrame you have to specify the field name to order by
      - It is specified as a tuple
    - The parameter named ascending allows to sort from the largest to the less number (True, by default) or viceversa (False)

Pandas: DataFrames operations

- The function apply allows one to apply a function over the columns or rows
- The function applymap allows one to apply a function element wise



```
In [110]: df = pd.DataFrame({'uno' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     ...:  'dos' : pd.Series([4,5, 6,7], index=['a', 'b', 'c', 'd']), 'tres' : pd.Series([8,9,10,11], index=['a',
'b', 'c', 'd'])})

In [111]: df
Out[111]:
   dos  tres  uno
a    4     8    1
b    5     9    2
c    6    10    3
d    7    11  NaN

In [112]: df.apply(lambda (x): x.max())
Out[112]:
dos      7
tres    11
uno      3
dtype: float64

In [113]: df.apply(lambda (x): [x.max(), x.min()])
Out[113]:
dos      [7.0, 4.0]
tres    [11.0, 8.0]
uno      [3.0, 1.0]
dtype: object

In [114]: df.apply(lambda (x): [x.max(), x.min()],axis=0)
Out[114]:
dos      [7.0, 4.0]
tres    [11.0, 8.0]
uno      [3.0, 1.0]
dtype: object

In [115]: df.apply(lambda (x): [x.max(), x.min()],axis=1)
Out[115]:
a     [8.0, 1.0]
b     [9.0, 2.0]
c    [10.0, 3.0]
d    [11.0, 7.0]
dtype: object
```



```
In [116]: df.applymap(lambda (x): (x, 2**x))
Out[116]:
        dos        tres         uno
a   (4, 16)    (8, 256)  (1.0, 2.0)
b   (5, 32)    (9, 512)  (2.0, 4.0)
c   (6, 64)  (10, 1024)  (3.0, 8.0)
d  (7, 128)  (11, 2048)  (nan, nan)

In [117]: df.applymap(lambda (x): [x.max(), x.min()])
Out[117]:
      dos      tres         uno
a  [4, 4]    [8, 8]  [1.0, 1.0]
b  [5, 5]    [9, 9]  [2.0, 2.0]
c  [6, 6]  [10, 10]  [3.0, 3.0]
d  [7, 7]  [11, 11]  [nan, nan]
```

DataFrames joining

- Like SQL's JOIN clause, pandas.merge allows two DataFrames to be joined on one or more keys
  - To specify the field to join by use the parameter *on*
- By default, pandas.merge operates as an *inner join*, which can be changed using the *how* parameter
  - how : {'left', 'right', 'outer', 'inner'}, default 'inner'
    - left: use only keys from left frame (SQL: left outer join)
    - right: use only keys from right frame (SQL: right outer join)
    - outer: use union of keys from both frames (SQL: full outer join)
    - inner: use intersection of keys from both frames (SQL: inner join)

DataFrames joining example

```
left_frame = pd.DataFrame({'key': range(5),
                           'left_value': ['a', 'b', 'c', 'd', 'e']})
right_frame = pd.DataFrame({'key': range(2, 7),
                           'right_value': ['f', 'g', 'h', 'i', 'j']})
```

```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e
```

```
   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```

inner join (default)

```
pd.merge(left_frame, right_frame, on='key', how='inner')
```

	key	left_value	right_value
0	2	c	f
1	3	d	g
2	4	e	h

left outer join

```
pd.merge(left_frame, right_frame, on='key', how='left')
```

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	f
3	3	d	g
4	4	e	h

right outer join

```
pd.merge(left_frame, right_frame, on='key', how='right')
```

	key	left_value	right_value
0	2	c	f
1	3	d	g
2	4	e	h
3	5	NaN	i
4	6	NaN	j

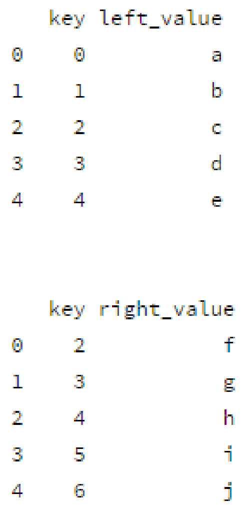
full outer join

```
pd.merge(left_frame, right_frame, on='key', how='outer')
```

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	f
3	3	d	g
4	4	e	h
5	5	NaN	i
6	6	NaN	j

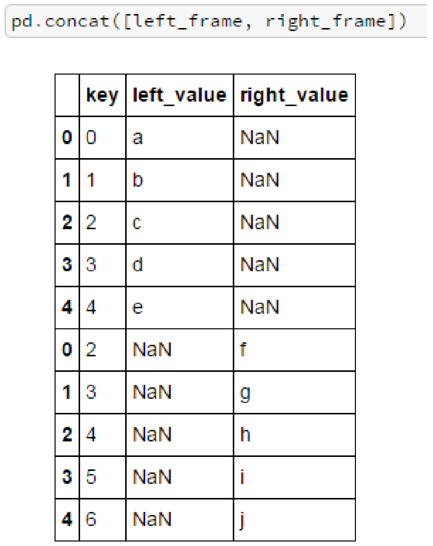
DataFrames combining

- Pandas also provides a way to combine DataFrames along an axis - pandas.concat
  - The function is equivalent to SQL's UNION clause



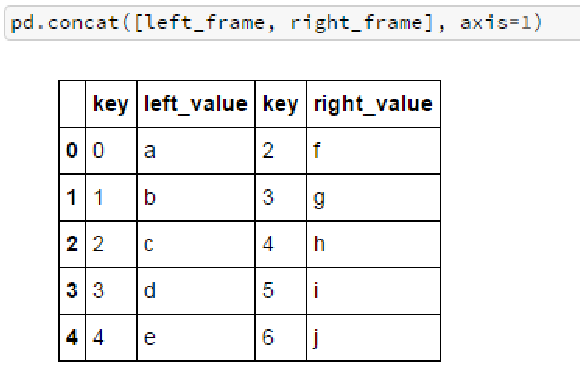
```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e
```

```
   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```



```
pd.concat([left_frame, right_frame])
```

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	NaN
3	3	d	NaN
4	4	e	NaN
0	2	NaN	f
1	3	NaN	g
2	4	NaN	h
3	5	NaN	i
4	6	NaN	j

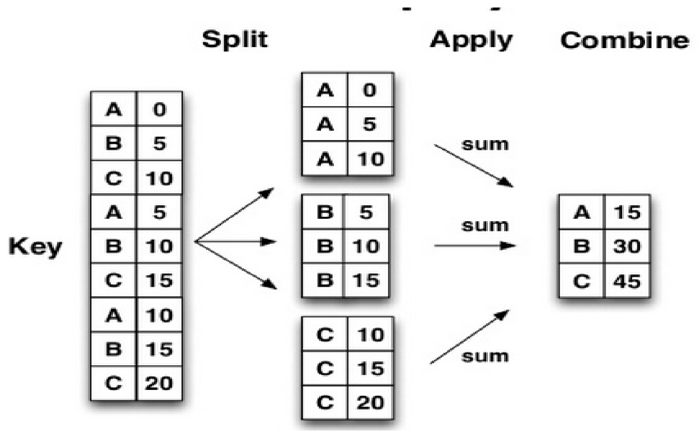


```
pd.concat([left_frame, right_frame], axis=1)
```

	key	left_value	key	right_value
0	0	a	2	f
1	1	b	3	g
2	2	c	4	h
3	3	d	5	i
4	4	e	6	j

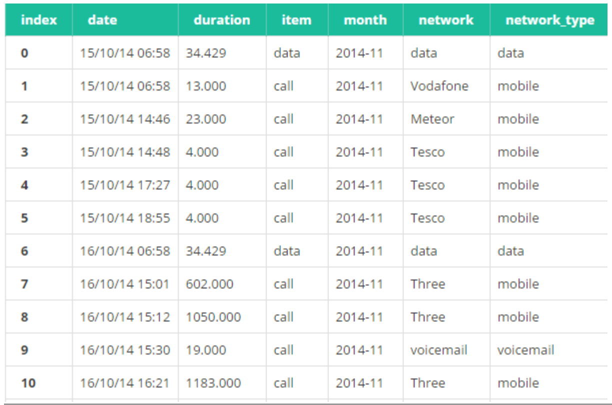
Pandas: DataFrame aggregation

- Groupby essentially splits the data into different groups depending on a variable of your choice
  - The groupby() function returns a GroupBy object, which have an attribute named groups that is a dictionary containing
    - Keys: computed unique groups
    - Values: the axis labels belonging to each group
    - These values can be accessed by the functions keys(), values() and items()
  - Functions like max(), min(), mean(), first(), last() can be quickly applied to the GroupBy object to obtain summary statistics for each group

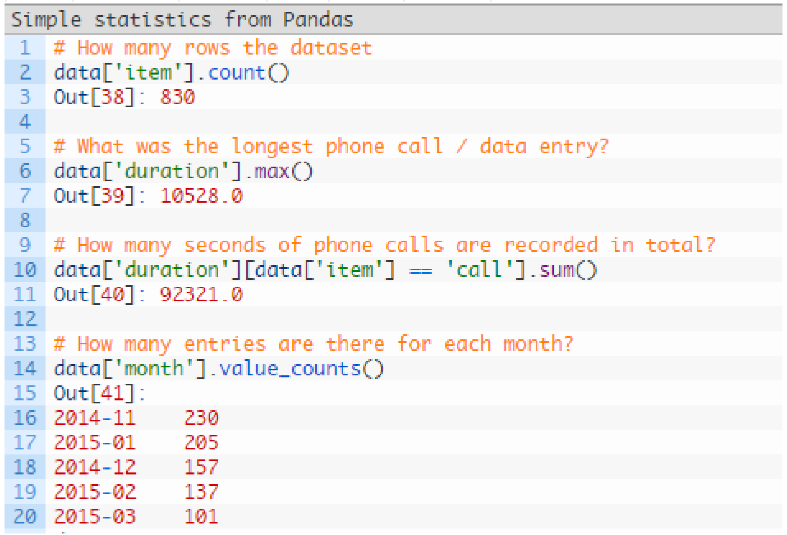


Pandas: DataFrame aggregation example

- Dataset contains 830 entries from my mobile phone log spanning a total time of 5 months

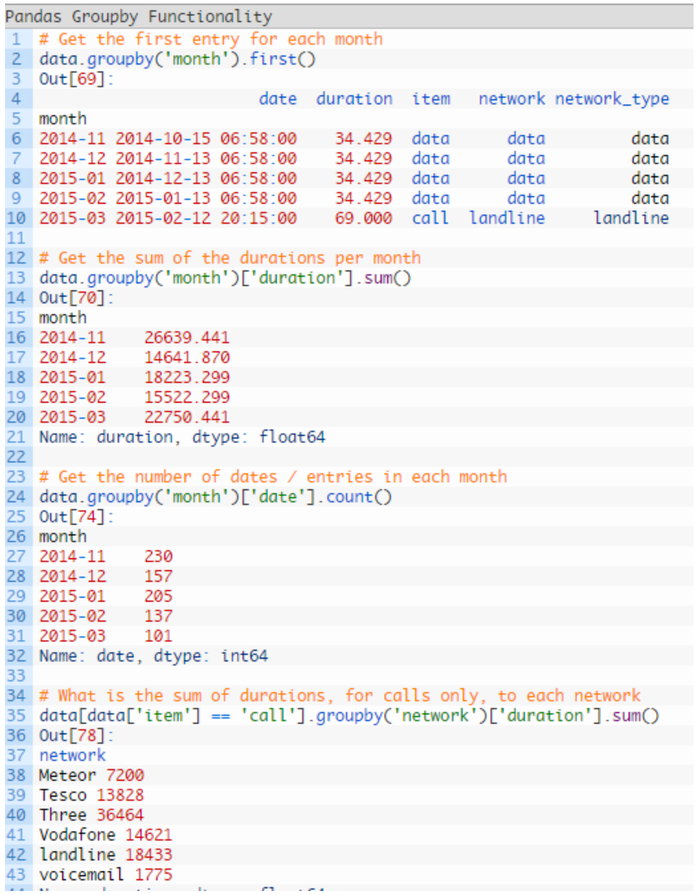


index	date	duration	item	month	network	network_type
0	15/10/14 06:58	34.429	data	2014-11	data	data
1	15/10/14 06:58	13.000	call	2014-11	Vodafone	mobile
2	15/10/14 14:46	23.000	call	2014-11	Meteor	mobile
3	15/10/14 14:48	4.000	call	2014-11	Tesco	mobile
4	15/10/14 17:27	4.000	call	2014-11	Tesco	mobile
5	15/10/14 18:55	4.000	call	2014-11	Tesco	mobile
6	16/10/14 06:58	34.429	data	2014-11	data	data
7	16/10/14 15:01	602.000	call	2014-11	Three	mobile
8	16/10/14 15:12	1050.000	call	2014-11	Three	mobile
9	16/10/14 15:30	19.000	call	2014-11	voicemail	voicemail
10	16/10/14 16:21	1183.000	call	2014-11	Three	mobile



Simple statistics from Pandas

```
# How many rows the dataset
data['item'].count()
Out[38]: 830

# What was the longest phone call / data entry?
data['duration'].max()
Out[39]: 10528.0

# How many seconds of phone calls are recorded in total?
data['duration'][data['item'] == 'call'].sum()
Out[40]: 92321.0

# How many entries are there for each month?
data['month'].value_counts()
Out[41]:
2014-11    230
2015-01    205
2014-12    157
2015-02    137
2015-03    101
```

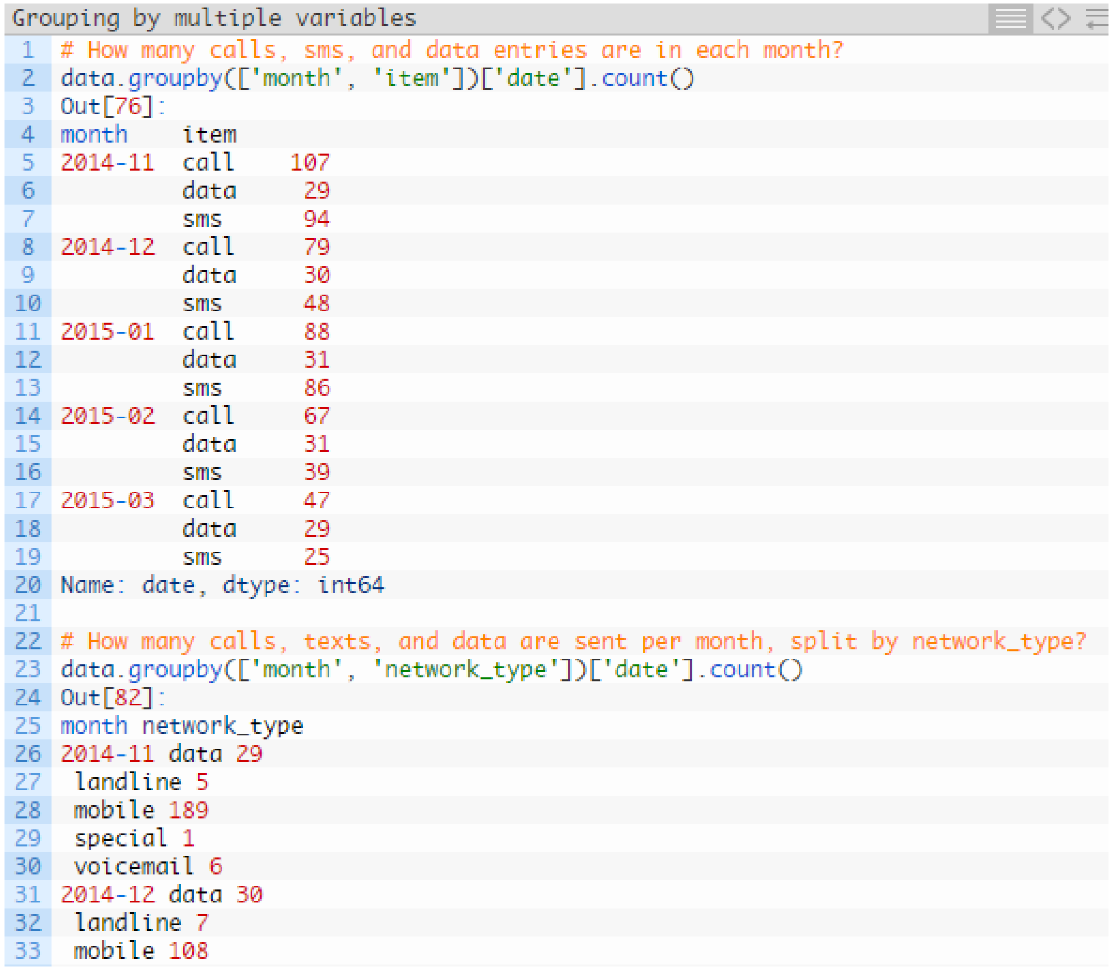


Pandas Groupby Functionality

```
# Get the first entry for each month
data.groupby('month').first()
Out[69]:
                        date  duration  item   network network_type
month
2014-11  2014-10-15 06:58:00    34.429  data      data         data
2014-12  2014-11-13 06:58:00    34.429  data      data         data
2015-01  2014-12-13 06:58:00    34.429  data      data         data
2015-02  2015-01-13 06:58:00    34.429  data      data         data
2015-03  2015-02-12 20:15:00    69.000  call  landline     landline

# Get the sum of the durations per month
data.groupby('month')['duration'].sum()
Out[70]:
month
2014-11    26639.441
2014-12    14641.870
2015-01    18223.299
2015-02    15522.299
2015-03    22750.441
Name: duration, dtype: float64

# Get the number of dates / entries in each month
data.groupby('month')['date'].count()
Out[74]:
month
2014-11    230
2014-12    157
2015-01    205
2015-02    137
2015-03    101
Name: date, dtype: int64

# What is the sum of durations, for calls only, to each network
data[data['item'] == 'call'].groupby('network')['duration'].sum()
Out[78]:
network
Meteor 7200
Tesco 13828
Three 36464
Vodafone 14621
landline 18433
voicemail 1775
```

Pandas: DataFrame aggregation example

- Data can be grouped by more than one variable

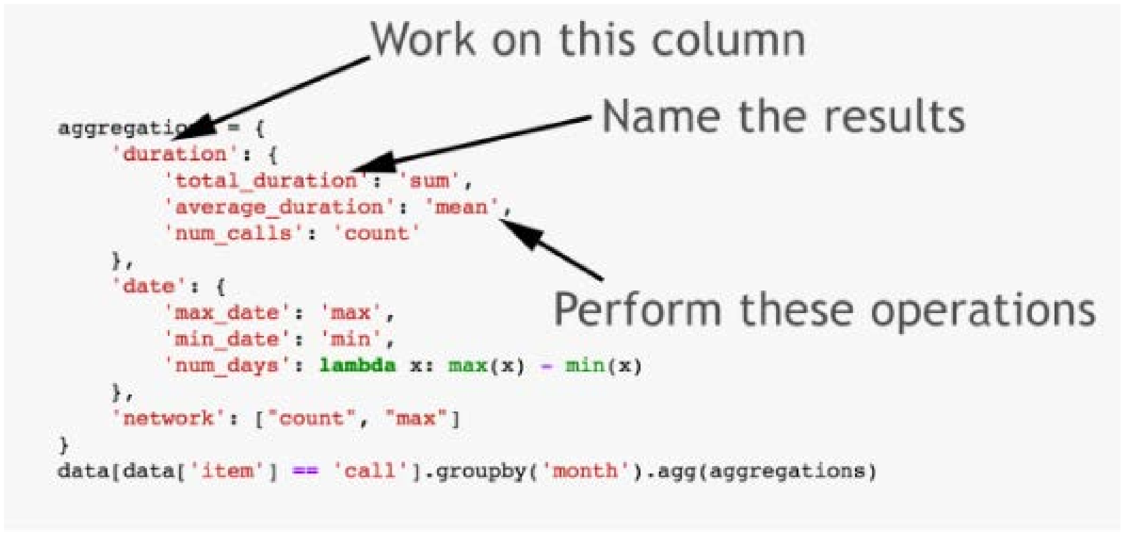


Grouping by multiple variables

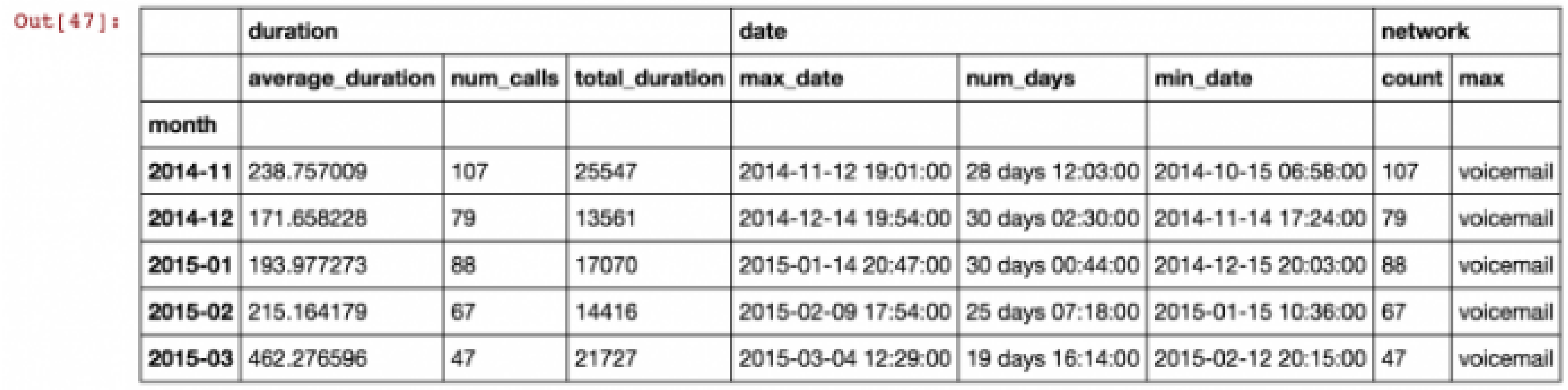
```
# How many calls, sms, and data entries are in each month?
data.groupby(['month', 'item'])['date'].count()
Out[76]:
month     item
2014-11   call     107
          data      29
          sms       94
2014-12   call      79
          data      30
          sms       48
2015-01   call      88
          data      31
          sms       86
2015-02   call      67
          data      31
          sms       39
2015-03   call      47
          data      29
          sms       25
Name: date, dtype: int64

# How many calls, texts, and data are sent per month, split by network_type?
data.groupby(['month', 'network_type'])['date'].count()
Out[82]:
month network_type
2014-11 data 29
 landline 5
 mobile 189
 special 1
 voicemail 6
2014-12 data 30
 landline 7
 mobile 108
```

Pandas: DataFrame aggregation

- Multiple Statistics per Group: agg() function



```
aggregations = {
    'duration': {
        'total_duration': 'sum',
        'average_duration': 'mean',
        'num_calls': 'count'
    },
    'date': {
        'max_date': 'max',
        'min_date': 'min',
        'num_days': lambda x: max(x) - min(x)
    },
    'network': ["count", "max"]
}
data[data['item'] == 'call'].groupby('month').agg(aggregations)
```



Out[47]:

	duration			date			network	
	average_duration	num_calls	total_duration	max_date	num_days	min_date	count	max
month								
2014-11	238.757009	107	25547	2014-11-12 19:01:00	28 days 12:03:00	2014-10-15 06:58:00	107	voicemail
2014-12	171.658228	79	13561	2014-12-14 19:54:00	30 days 02:30:00	2014-11-14 17:24:00	79	voicemail
2015-01	193.977273	88	17070	2015-01-14 20:47:00	30 days 00:44:00	2014-12-15 20:03:00	88	voicemail
2015-02	215.164179	67	14416	2015-02-09 17:54:00	25 days 07:18:00	2015-01-15 10:36:00	67	voicemail
2015-03	462.276596	47	21727	2015-03-04 12:29:00	19 days 16:14:00	2015-02-12 20:15:00	47	voicemail

Pandas: DataFrame aggregation

- The agg() function returns MultiIndexes
  - In the previous example
    - (‘duration‘, ‘total_duration‘), (‘duration‘, ‘average_duration‘), (‘duration‘, ‘num_calls‘)
    - (‘date‘, ‘max_date‘), (‘date‘, ‘min_date‘), (‘date‘, ‘num_days‘)
    - (‘network‘, ‘count‘), (‘network‘, ‘max‘)
- If you want to operate over the result of the agg function
  - You have to specify the column by the corresponding MultiIndex
    - Example: data.sort_values((‘network’, ‘count’), ascending=False)

Pandas: visualization

- Pandas visualization uses the matplotlib package
  - import matplotlib.pyplot as plt
- The plot method on Series and DataFrame is just a simple wrapper around plt.plot()
- Basic plotting: plot()

```
In [2]: ts = pd.Series(np.random.randn(1000),
index=pd.date_range('1/1/2000', periods=1000))

In [3]: ts = ts.cumsum()

In [4]: ts.plot()
```

```
In [5]: df =
pd.DataFrame(np.random.randn(1000, 4),
index=ts.index, columns=list('ABCD'))

In [6]: df = df.cumsum()

In [7]: plt.figure(); df.plot();
```

Pandas: visualization

- Other options: DataFrame.plot.<kind>
  - df.plot.area
  - df.plot.bar (df.plot.barh in horizontal)
  - df.plot.density
  - df.plot.hist
  - df.plot.line
  - df.plot.scatter
  - df.plot.box
  - df.plot.hexbin
  - df.plot.kde
  - df.plot.pie
- See the documentation for details

More information about pandas

- http://pandas.pydata.org/pandas-docs/version/0.18.0/pandas.pdf