# Network Measurement and Monitoring
# Pandas

Y. Vanaubel, B. Donnet

## 1 Introduction

This document is an introduction to PANDAS[1], an an open-source Python library that allows you to manipulate data and compute statistics on it.

You will first learn how to store data in specific structures. Then, you will see how you can interact with these structures, as for example, sorting the content or getting and setting some values. Finally, you will be shown different statistic tools which are useful to describe your dataset correctly, as the mean, quantiles, histograms, and so on.

## 2 Importing Pandas

PANDAS requires the NUMPY library in order to work properly.[2] Both libraries can be imported in your script as following:

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

## 3 Data Structures

These structures provide a convenient way to manipulate your data. In this lab, we will study two main structures: `Series` and `DataFrame`.

### 3.1 `Series`

This structure is a one-dimensional labeled array which can contain any type of data (integers, strings, floating point numbers, Python objects, etc.). In a `Series`, the axis labels[3] are named `index`.

A `Series` can be created using the following command:

[1] `https://pandas.pydata.org/`
[2] NUMPY is the fundamental package for scientific computing with Python: `http://www.numpy.org/`
[3] PANDAS does not require labels to be unique in a structure.

```
s = pd.Series(data, index=index, name=name)
```

where *index* is a list of axis labels, *name* is a name given to the `Series`, and *data* can be:

- an `ndarray`[4]
- a Python `dict`
- a scalar value

Note that *index* and *name* arguments are optional.

### a. ndarray

In the case of an `ndarray`, *index* must have the same length as *data*. If the labels are not provided, default labels are created with the values $[0, 1, 2, ..., len(data) - 1]$.

```
# Create an ndarray with integers 1, 2 and 3
>>> a = np.array([1, 2, 3])

# Create a Series from the array
>>> pd.Series(a, index=['a', 'b', 'c'])
a     1
b     2
c     3
dtype: int64

# Create another Series with a name and no labels
>>> pd.Series(a, name="excitingseries")
0     1
1     2
2     3
Name: excitingseries, dtype: int64
```

### b. dict

If you build a `Series` from a `dict`, the sorted keys of the dictionary will be used to generate the index, if possible.

```
# Build a dictionary with float values
>>> d = {'a' : 3.1, 'b' : 1., 'c' : 2.}
>>> d
{'a': 3.1, 'c': 2.0, 'b': 1.0}
```

[4]See the NumPy library for more information.

```
# Create the Series from the dictionary
>>> pd.Series(d)
a    3.1
b    1.0
c    2.0
dtype: float64
```

If *index* is given when creating the `Series`, Pandas will select the values in *data* that correspond to the labels in *index*.

```
# Create the Series from the previous dictionary,
# specifying the labels to select
>>> pd.Series(d, index=['b', 'd', 'a'])
b    1.0
d    NaN
a    3.1
dtype: float64
```

#### c. scalar value

Finally, if you create a `Series` based on a scalar value, an *index* must be provided. Each label will then be associated to the scalar value.

```
>>> pd.Series(8., index=['a', 'b', 'd'])
a    8.0
b    8.0
d    8.0
dtype: float64
```

### 3.2 `DataFrame`

The `DataFrame` is the most commonly used object in Pandas. It is a 2-dimensional labeled data structure. It can be seen as a SQL table or a dictionary of `Series` objects. Its columns may be of different types.

When creating a `DataFrame`, you can specify the row labels via the *index* argument, and the column labels using the *columns* argument. These two arguments are optional, but using them may ensure the index and/or columns of the resulting `DataFrame`. For example, when creating a `DataFrame` based on a `dict` of `Series`, the data not matching the labels in *index* will be discarded.

Note that if axis labels are not given, they are built from the input data based on common sense rules.

As the `Series`, a `DataFrame` can be build based on different input objects:

- a `dict` of 1-dimensional `ndarrays`, `lists`, `objects`, or `Series`

- a 2-dimensional `ndarray`
- a structured or record `ndarray`[5]
- a `Series`
- another `DataFrame`

### a. `dict` of `Series` or `objects`

If the `DataFrame` is built based on a dictionary of `Series`, the resulting *index* is the union of the indexes of the different `Series`. If the *columns* are not specified, the resulting *columns* will be the sorted list of the `dict` keys.

```
# Create a dict of two series
>>> d = {'one' : pd.Series([1, 2], index=['a', 'b']),
...      'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}

# Create the DataFrame based on the dict
>>> pd.DataFrame(d)
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

# Create a DataFrame selecting 2 indexes and 2 columns
>>> df = pd.DataFrame(d, index=['c', 'a'], columns=['two', 'three'])
>>> df
   two three
c  3.0   NaN
a  1.0   NaN
```

If the `DataFrame` is built from a dictionary of `objects`, the different `objects` are converted into `Series`, if possible.

```
>>> df = pd.DataFrame({ 'A' : 1.,
...                     'B' : pd.date_range('20171124', periods=4),
...                     'C' : pd.Series(1,index=[1,8,7,9]),
...                     'D' : np.array([3] * 4,dtype='int32'),
...                     'E' : ["Benoit","likes","good","grades"],
...                     'F' : 'foo' })
>>> df
     A          B  C  D       E    F
1  1.0 2017-11-24  1  3  Benoit  foo
8  1.0 2017-11-25  1  3   likes  foo
7  1.0 2017-11-26  1  3    good  foo
9  1.0 2017-11-27  1  3  grades  foo
```

[5] Not covered here, see documentation for more details.

Note that in this example, the function `date_range()` returns a `DatetimeIndex`. For more information, see PANDAS' documentation.

### b. `dict` of `ndarrays` or `lists`

In this case, the `ndarrays` or `lists` must have the same length. If *index* is specified, its length must also be the same as the arrays/lists. In the other case, the labels will be created such as `range(n)` where `n` is the length of the arrays/lists.

```
# Create a dict of two lists
>>> d = {'one' : [1, 2, 3, 4], 'two' : [4, 3, 2, 1]}

# Create the DataFrame
>>> pd.DataFrame(d)
   one  two
0    1    4
1    2    3
2    3    2
3    4    1
```

### c. `list` of `dicts`

```
# Create a list of dictionaries
>>> d = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

# Create a DataFrame from the list
>>> pd.DataFrame(d)
   a   b     c
0  1   2   NaN
1  5  10  20.0

# Specify the indexes
>>> pd.DataFrame(d, index=['first', 'second'])
        a   b     c
first   1   2   NaN
second  5  10  20.0

# Select some columns
>>> pd.DataFrame(d, columns=['a', 'c'])
   a     c
0  1   NaN
1  5  20.0
```

**d.** `Series`

In this case, the `DataFrame` will have the same *index* as the `Series`, and a single column having the name of the `Series` as label, if any.

```
# Create a Series
>>> s = pd.Series(8., index=['a', 'b', 'd'], name="magic")

# Create the DataFrame
>>> pd.DataFrame(s)
   magic
a    8.0
b    8.0
d    8.0

# Create a Series without any name
>>> s = pd.Series(8., index=['a', 'b', 'd'])

# Create a DataFrame
>>> pd.DataFrame(s)
     0
a  8.0
b  8.0
d  8.0

# Create the DataFrame specifying a column name
>>> pd.DataFrame(s, columns=['one'])
   one
a  8.0
b  8.0
d  8.0
```

# 4 Missing Data

You can construct structures with missing data. To specify a missing value, you can use the "not a number" expression, available in the NumPy library: `np.nan`.

```
>>> d = {'one' : [1, np.nan, 3, 4], 'two' : [4, 3, 2, np.nan]}
>>> pd.DataFrame(d)
   one  two
0  1.0  4.0
1  NaN  3.0
2  3.0  2.0
3  4.0  NaN
```

Note that missing data may appear when computing some operations on the structures, like statistical manipulations. You can detect these values using the functions `isna()` and `notna()`, which are also methods of `Series` and `DataFrame` objects.

```
# Build a Series with missing values
>>> d = {'a' : 3.1, 'b' : np.nan, 'c' : 2., 'd' : np.nan}
>>> s = pd.Series(d)
>>> s
a    3.1
b    NaN
c    2.0
d    NaN
dtype: float64

# Find indexes with NaN values
>>> pd.isna(s)
a    False
b     True
c    False
d     True
dtype: bool
# Same effect as previous function
>>> s.isna()
a    False
b     True
c    False
d     True
dtype: bool

# Find indexes with non-missing values
>>> s.notna()
a     True
b    False
c     True
d    False
dtype: bool
```

You can easily drop the data with missing values, or replace these missing values.

```
>>> d2 = {'a' : 1.2, 'b' : 3, 'c' : 8.3, 'd' : np.nan}
>>> df = pd.DataFrame({'one' : d, 'two' : d2})
>>> df
   one  two
a  3.1  1.2
b  NaN  3.0
c  2.0  8.3
d  NaN  NaN

# Drop rows containing at least one missing data
>>> df.dropna(how='any')
   one  two
a  3.1  1.2
c  2.0  8.3
```

```
# Drop rows containing only NaNs
>>> df.dropna(how='all')
   one  two
a  3.1  1.2
b  NaN  3.0
c  2.0  8.3

# Replace any missing data with 10
>>> df.fillna(value=10)
    one   two
a   3.1   1.2
b  10.0   3.0
c   2.0   8.3
d  10.0  10.0
```

# 5 Data Access and Setting

This section describes how you can manipulate the data structures in order to set, access or isolate some information, in order to compute some statistics on it later.

We will use the following `DataFrame`[6] in the different examples:

```
>>> df = pd.DataFrame({'A' : pd.date_range('20171124', periods=4),
...                    'B' : pd.Series([8,8,1,1],index=[1,3,5,7]),
...                    'C' : ["one","four","four","five"],
...                    'D' : np.array([6.] * 4,dtype='float32')})
>>> df
           A  B     C    D
1 2017-11-24  8   one  6.0
3 2017-11-25  8  four  6.0
5 2017-11-26  1  four  6.0
7 2017-11-27  1  five  6.0
```

## 5.1 Getting Data

The row and column labels can be retrieved by accessing respectively the *index* and *columns* attributes. The underlying NumPy data is available thanks to the *values* attribute, while the objects' types can be obtained via the *dtypes* attribute.

```
# Get the labels for the rows and columns
>>> df.index
Int64Index([1, 3, 5, 7], dtype='int64')
>>> df.columns
Index(['A', 'B', 'C', 'D'], dtype='object')
```

[6]The concepts presented in this section also apply to the `Series`.

```
# Get the data
>>> df.values
array([[Timestamp('2017-11-24 00:00:00'), 8, 'one', 6.0],
       [Timestamp('2017-11-25 00:00:00'), 8, 'four', 6.0],
       [Timestamp('2017-11-26 00:00:00'), 1, 'four', 6.0],
       [Timestamp('2017-11-27 00:00:00'), 1, 'five', 6.0]],
      dtype=object)

# Get the objects' types
>>> df.dtypes
A        datetime64
B             int64
C            object
D           float32
dtype: object
```

### 5.2 Data Selection

#### a. Columns

A `DataFrame` column can be selected quite easily, the result being a `Series` with the column's label as name.

```
# Select the column A
>>> df['A']
1   2017-11-24
3   2017-11-25
5   2017-11-26
7   2017-11-27
Name: A, dtype: datetime64
# Another manner to select the column
>>> df.A
1   2017-11-24
3   2017-11-25
5   2017-11-26
7   2017-11-27
Name: A, dtype: datetime64
```

You can select multiple columns using a list of labels.

```
# Select the columns A and D
>>> df[['A','D']]
           A    D
1 2017-11-24  6.0
3 2017-11-25  6.0
5 2017-11-26  6.0
7 2017-11-27  6.0
```

### b. Rows

The top and bottom rows of a structure can be accessed easily thanks to `head()` and `tail()` methods.

```
# Get the three first rows
>>> df.head(3)
            A  B     C    D
1 2017-11-24  8   one  6.0
3 2017-11-25  8  four  6.0
5 2017-11-26  1  four  6.0

# Get the two last rows
>>> df.tail(2)
            A  B     C    D
5 2017-11-26  1  four  6.0
7 2017-11-27  1  five  6.0
```

You can select specific rows using *loc* and index names.

```
# Get the row with label 5
>>> df.loc[5]
A    2017-11-26 00:00:00
B                      1
C                   four
D                      6
Name: 5, dtype: object

# Get the rows with labels 5 and 1
# Here, a list of labels is needed
>>> df.loc[[5,1]]
            A  B     C    D
5 2017-11-26  1  four  6.0
1 2017-11-24  8   one  6.0
```

If you need to select multiple consecutive rows, you can use the slicing.

```
# Select rows 0 to 2 (indexing starts at 0)
# Note the the endpoint is not included
>>> df[0:3]
            A  B     C    D
1 2017-11-24  8   one  6.0
3 2017-11-25  8  four  6.0
5 2017-11-26  1  four  6.0
```

The selection by position is also allowed using *iloc*.

```
# Select row at position 2
# Note the resulting object is a Series
>>> df.iloc[2]
A     2017-11-26 00:00:00
B                       1
C                    four
D                       6
Name: 5, dtype: object
```

You can slice with *iloc*, or specify a list of positions.

```
# Select rows at positions 2 to 3
>>> df.iloc[2:4]
           A  B     C    D
5 2017-11-26  1  four  6.0
7 2017-11-27  1  five  6.0

# Select rows at positions 1 and 3
>>> df.iloc[[1,3]]
           A  B     C    D
3 2017-11-25  8  four  6.0
7 2017-11-27  1  five  6.0
```

### c. Multiple Axes

You can select on multiple axes by labels using *loc* and the slicing.

```
 # Get the columns A and C
>>> df.loc[:,['A','C']]
           A     C
1 2017-11-24   one
3 2017-11-25  four
5 2017-11-26  four
7 2017-11-27  five

# Get columns A and C for rows with labels between 1 and 5
# Note that both endpoint labels are included
>>> df.loc[1:5,['A','C']]
           A     C
1 2017-11-24   one
3 2017-11-25  four
5 2017-11-26  four
```

```
# Get columns A and C for row with label 3
# Note that you obtain a Series
>>> df.loc[3,['A','C']]
A    2017-11-25 00:00:00
C                   four
Name: 3, dtype: object
```

The selection by position is also allowed using *iloc*.

```
# Select the two first columns on the 3 last rows
>>> df.iloc[1:4,0:2]
           A  B
3 2017-11-25  8
5 2017-11-26  1
7 2017-11-27  1

# Select all rows for the two first columns
>>> df.iloc[:,0:2]
           A  B
1 2017-11-24  8
3 2017-11-25  8
5 2017-11-26  1
7 2017-11-27  1

# Select all columns for the two first rows
>>> df.iloc[0:2,:]
           A  B     C    D
1 2017-11-24  8   one  6.0
3 2017-11-25  8  four  6.0
```

### d. Single Element

You can retrieve a single value from the `DataFrame` with *at* and labels.

```
>>> df.at[3, 'A']
Timestamp('2017-11-25 00:00:00')

>>> df.at[5, 'C']
'four'
```

Similarly to *iloc*, you can use *iat* to get access to an element by position.

```
>>> df.iat[1,2]
'four'
```

### 5.3 Selection with Conditions

With PANDAS, you are allowed to select data that meets a boolean condition.

```
# Select rows having the value in column B lower than 5
>>> df[df.B < 5]
           A  B     C    D
5 2017-11-26  1  four  6.0
7 2017-11-27  1  five  6.0

# Select rows with a date being 2017-11-25 or after
>>> df[df.A >= '20171125']
           A  B     C    D
3 2017-11-25  8  four  6.0
5 2017-11-26  1  four  6.0
7 2017-11-27  1  five  6.0
```

You can also select values that meet a condition.

```
>>> df[df > 6]
           A    B     C    D
1 2017-11-24  8.0   one  NaN
3 2017-11-25  8.0  four  NaN
5 2017-11-26  NaN  four  NaN
7 2017-11-27  NaN  five  NaN
```

The method `isin()` allows you to filter based on specific values.

```
>>> df[df.C.isin(['four', 'five'])]
           A  B     C    D
3 2017-11-25  8  four  6.0
5 2017-11-26  1  four  6.0
7 2017-11-27  1  five  6.0
```

### 5.4 Data Setting

The different selection methods presented previously can be used to set values in the structures.

```
# Copy df
>>> df2 = df.copy()

# Replace the last column
>>> s = pd.Series([5.,4.,4.,5.], index=[1,3,5,7])
>>> df2['D'] = s
```

```
# Change the element at line with label 5 and column with label B
>>> df2.at[5,'B'] = 2

# Change the element at position [0,2]
>>> df2.iat[0,2] = "three"

# Add a new column
>>> df2.loc[:,'E'] = np.array([7] * len(df2))

# Display the result
>>> df2
            A  B      C    D  E
1 2017-11-24  8  three  5.0  7
3 2017-11-25  8   four  4.0  7
5 2017-11-26  2   four  4.0  7
7 2017-11-27  1   five  5.0  7
```

## 6 Operations

This section describes some operations that can be applied on the data structures.

Generally, an operation excludes missing data, and can take an axis as argument which can be specified by name or integer:

- "index", axis=0, default
- "columns", axis=1

The operation is then realized alongside the specified axis.

We will use the following `DataFrame`[7] in the different examples:

```
>>> dates = pd.date_range('20180101', periods=6, freq="M")
>>> df = pd.DataFrame(np.random.randn(6,4),
...                   index=dates,
...                   columns=['A','B','C','D'])
>>> df.iloc[0,3] = np.nan
>>> df.iloc[2,1] = np.nan
>>> df.iloc[2,3] = np.nan

>>> df
                   A         B         C         D
2018-01-31 -0.036007  2.425324 -0.160817       NaN
2018-02-28  1.458603 -0.687165  1.458743  0.234733
2018-03-31 -0.860041       NaN -0.138422       NaN
2018-04-30 -1.082399  0.274795  0.935195  0.939475
2018-05-31 -1.386264  0.845519  2.252664  0.216155
2018-06-30 -0.926516  0.049159  0.094237  1.341607
```

[7]The concepts presented in this section also apply to the `Series`.

## 6.1 Basic Operations

These simple operations are done thanks to the methods `add()`, `sub()`, `mul()`, and `div()`.

```
# Add 3 to each element
>>> df+3
                   A          B          C          D
2018-01-31  2.963993  5.425324  2.839183        NaN
2018-02-28  4.458603  2.312835  4.458743  3.234733
2018-03-31  2.139959        NaN  2.861578        NaN
2018-04-30  1.917601  3.274795  3.935195  3.939475
2018-05-31  1.613736  3.845519  5.252664  3.216155
2018-06-30  2.073484  3.049159  3.094237  4.341607

# Create a Series
>>> s = pd.Series([1,3,5,np.nan,6,8], index=dates)
>>> s
2018-01-31    1.0
2018-02-28    3.0
2018-03-31    5.0
2018-04-30    NaN
2018-05-31    6.0
2018-06-30    8.0
Freq: M, dtype: float64

# Subtract the values in the Series to each column
>>> df.sub(s, axis=0)
                   A          B          C          D
2018-01-31 -1.036007  1.425324 -1.160817        NaN
2018-02-28 -1.541397 -3.687165 -1.541257 -2.765267
2018-03-31 -5.860041        NaN -5.138422        NaN
2018-04-30       NaN        NaN        NaN        NaN
2018-05-31 -7.386264 -5.154481 -3.747336 -5.783845
2018-06-30 -8.926516 -7.950841 -7.905763 -6.658393

# Multiply the 4 first values in the Series with each row
>>> df.mul(s.values[:4], axis=1)
                   A          B          C   D
2018-01-31 -0.036007  7.275971  -0.804084 NaN
2018-02-28  1.458603 -2.061496   7.293714 NaN
2018-03-31 -0.860041        NaN  -0.692110 NaN
2018-04-30 -1.082399  0.824386   4.675975 NaN
2018-05-31 -1.386264  2.536556  11.263321 NaN
2018-06-30 -0.926516  0.147477   0.471187 NaN
```

## 6.2 Data Sorting

The data can be sorted based on the labels.

```
# Sort on the rows, descending
>>> df.sort_index(axis=0, ascending=False)
                   A         B         C         D
2018-06-30 -0.926516  0.049159  0.094237  1.341607
2018-05-31 -1.386264  0.845519  2.252664  0.216155
2018-04-30 -1.082399  0.274795  0.935195  0.939475
2018-03-31 -0.860041       NaN -0.138422       NaN
2018-02-28  1.458603 -0.687165  1.458743  0.234733
2018-01-31 -0.036007  2.425324 -0.160817       NaN

# Sort on the columns, descending
>>> df.sort_index(axis=1, ascending=False)
                   D         C         B         A
2018-01-31       NaN -0.160817  2.425324 -0.036007
2018-02-28  0.234733  1.458743 -0.687165  1.458603
2018-03-31       NaN -0.138422       NaN -0.860041
2018-04-30  0.939475  0.935195  0.274795 -1.082399
2018-05-31  0.216155  2.252664  0.845519 -1.386264
2018-06-30  1.341607  0.094237  0.049159 -0.926516
```

The data can also be sorted based on the values.

```
# Sort on column D
>>> df.sort_values(by='D')
                   A         B         C         D
2018-05-31 -1.386264  0.845519  2.252664  0.216155
2018-02-28  1.458603 -0.687165  1.458743  0.234733
2018-04-30 -1.082399  0.274795  0.935195  0.939475
2018-06-30 -0.926516  0.049159  0.094237  1.341607
2018-01-31 -0.036007  2.425324 -0.160817       NaN
2018-03-31 -0.860041       NaN -0.138422       NaN

# Sort on columns D and A
>>> df.sort_values(by=['D', 'A'])
                   A         B         C         D
2018-05-31 -1.386264  0.845519  2.252664  0.216155
2018-02-28  1.458603 -0.687165  1.458743  0.234733
2018-04-30 -1.082399  0.274795  0.935195  0.939475
2018-06-30 -0.926516  0.049159  0.094237  1.341607
2018-03-31 -0.860041       NaN -0.138422       NaN
2018-01-31 -0.036007  2.425324 -0.160817       NaN
```

### 6.3 Value Counts

The method `value_counts()` counts the different values appearing in a `Series`.[8]

[8] This method does not apply on a `DataFrame`.

```
# Create a Series with random values between 0 and 9
>>> data = np.random.randint(0, 10, size=50)
>>> s = pd.Series(data)

# Count the values
>>> s.value_counts()
1    12
9     7
7     7
4     6
8     4
6     3
3     3
2     3
0     3
5     2
dtype: int64
```

### 6.4 Data Transposition

The data can be transposed using *T*.

```
# Get the four first rows and transpose the resulting DataFrame
>>> df[:4].T
   2018-01-31  2018-02-28  2018-03-31  2018-04-30
A   -0.036007    1.458603   -0.860041   -1.082399
B    2.425324   -0.687165         NaN    0.274795
C   -0.160817    1.458743   -0.138422    0.935195
D         NaN    0.234733         NaN    0.939475
```

### 6.5 Descriptive Statistics

Different statistical operations can be applied on a data structure. A list of common methods is available in Table 1.

The methods have a *skipna* option specifying if the missing data must be discarded or not (*True* by default).

Some examples are given hereafter.

```
# Sum the values in the columns, including NaN
>>> df.sum(0, skipna=False)
A   -2.832625
B         NaN
C    4.441601
D         NaN
dtype: float64
```

| Method | Description |
|---|---|
| `count` | Number of non-NA observations |
| `sum` | Sum of values |
| `mean` | Mean of values |
| `mad` | Mean absolute deviation |
| `median` | Arithmetic median of values |
| `min` | Minimum |
| `max` | Maximum |
| `abs` | Absolute value |
| `prod` | Product of values |
| `std` | Bessel-corrected sample standard deviation |
| `var` | Unbiased variance |
| `sem` | Standard error of the mean |
| `quantile` | Sample quantile (value at %) |
| `cumsum` | Cumulative sum |
| `cumprod` | Cumulative product |
| `cummax` | Cumulative maximum |
| `cummin` | Cumulative minimum |

Table 1 – Common statistical methods.

```
# Mean on the columns
>>> df.mean()
A   -0.472104
B    0.581526
C    0.740267
D    0.682993
dtype: float64

# Mean for each rows
>>> df.mean(1)
2018-01-31    0.742833
2018-02-28    0.616228
2018-03-31   -0.499232
2018-04-30    0.266767
2018-05-31    0.482018
2018-06-30    0.139622
Freq: M, dtype: float64

# Standard deviation for each row
>>> df.std(1)
2018-01-31    1.458415
2018-02-28    1.043040
2018-03-31    0.510262
2018-04-30    0.952128
2018-05-31    1.508695
2018-06-30    0.929455
Freq: M, dtype: float64
```

```
# Cumulative sum on the columns
>>> df.cumsum()
                   A         B         C         D
2018-01-31 -0.036007  2.425324 -0.160817       NaN
2018-02-28  1.422595  1.738159  1.297926  0.234733
2018-03-31  0.562554       NaN  1.159504       NaN
2018-04-30 -0.519844  2.012954  2.094699  1.174209
2018-05-31 -1.906109  2.858473  4.347363  1.390363
2018-06-30 -2.832625  2.907632  4.441601  2.731970
```

You can obtain a variety of summary statistics about a `Series` or the columns of a `DataFrame` thanks to the `describes()` method (excluding missing values).

```
>>> df.describe()
              A         B         C         D
count  6.000000  5.000000  6.000000  4.000000
mean  -0.472104  0.581526  0.740267  0.682993
std    1.047181  1.167942  0.984170  0.553302
min   -1.386264 -0.687165 -0.160817  0.216155
25%   -1.043428  0.049159 -0.080257  0.230089
50%   -0.893279  0.274795  0.514716  0.587104
75%   -0.242016  0.845519  1.327856  1.040008
max    1.458603  2.425324  2.252664  1.341607

# Specifies the percentiles to display
>>> df.describe(percentiles=[.10, .30, .60, .90])
              A         B         C         D
count  6.000000  5.000000  6.000000  4.000000
mean  -0.472104  0.581526  0.740267  0.682993
std    1.047181  1.167942  0.984170  0.553302
min   -1.386264 -0.687165 -0.160817  0.216155
10%   -1.234331 -0.392636 -0.149619  0.221728
30%   -1.004458  0.094286 -0.022092  0.232875
50%   -0.893279  0.274795  0.514716  0.587104
60%   -0.860041  0.503085  0.935195  0.798527
90%    0.711298  1.793402  1.855704  1.220967
max    1.458603  2.425324  2.252664  1.341607
```

For non-numerical content, the method will determine the number of unique values and most frequently occurring values. For mixed-type content, only the numerical columns will be considered. This behavior can be modified (see documentation).

```
>>> s = pd.Series(['b', 'a', 'n', 'a', 'n', 'a', np.nan])
>>> s.describe()
count      6
unique     3
top        a
freq       3
dtype: object
```

## 6.6 Row and Column-wise Function Application

The `apply()` function allows to apply arbitrary functions along the axes of a `DataFrame`.

```
# Compute the mean on the different rows
>>> df.apply(np.mean, axis=1)
2018-01-31     0.742833
2018-02-28     0.616228
2018-03-31    -0.499232
2018-04-30     0.266767
2018-05-31     0.482018
2018-06-30     0.139622
Freq: M, dtype: float64

# Get the difference between max and min values on each column
>>> df.apply(lambda x: x.max() - x.min())
A     2.844867
B     3.112489
C     2.413481
D     1.125452
dtype: float64
```

## 6.7 Data Grouping

Grouping elements in structures allows you to group the data into groups based on specific criteria, apply a function to each group independently, and combine the results in a structure.

```
# Create a new dataframe
>>> df2 = pd.DataFrame({'A' : ['one','two','one','two',
...                            'one','one','two','two'],
...                     'B' : ['red','red','blue','red',
...                            'blue','red','blue','blue'],
...                     'C' : np.random.randint(0, 10, size=8),
...                     'D' : np.random.randint(0, 10, size=8)})

>>> df2
     A     B  C  D
0  one   red  3  1
1  two   red  8  2
2  one  blue  3  4
3  two   red  0  5
4  one  blue  4  2
5  one   red  8  9
6  two  blue  5  1
7  two  blue  9  3
```

```
# Group on the first column and sum the values in each group.
>>> df2.groupby('A').sum()
      C   D
A
one  18  16
two  22  11

# Group on multiple columns
>>> df3 = df2.groupby(['A','B']).sum()
>>> df3
           C   D
A   B
one blue   7   6
    red   11  10
two blue  14   4
    red    8   7

# Access the columns with indexes one and red
>>> df3.loc['one','red']
C    11
D    10
Name: (one, red), dtype: int64
```

## 6.8 Merging Structures

You can concatenate structures on both axes with the method `concat()`. You can also use the method `append()` which will concatenate rows only. Note that when concatenating rows, the indexes must be disjoint, while it is not necessary for the column labels.

```
# Create another dataframe
>>> dates2 = pd.date_range('20180701', periods=2, freq="M")
>>> df2 = pd.DataFrame(np.random.randn(2,3),
...                    index=dates2,
...                    columns=['A', 'B','C'])
>>> df2
                   A         B         C
2018-07-31  0.904905 -1.934648  1.325258
2018-08-31 -0.107061 -0.151684 -0.527963

# Append df and df2
>>> df.append(df2)
                   A         B         C         D
2018-01-31 -0.036007  2.425324 -0.160817       NaN
2018-02-28  1.458603 -0.687165  1.458743  0.234733
2018-03-31 -0.860041       NaN -0.138422       NaN
2018-04-30 -1.082399  0.274795  0.935195  0.939475
2018-05-31 -1.386264  0.845519  2.252664  0.216155
2018-06-30 -0.926516  0.049159  0.094237  1.341607
2018-07-31  0.904905 -1.934648  1.325258       NaN
2018-08-31 -0.107061 -0.151684 -0.527963       NaN
```

```
# Add a new column to df, identical to column A
>>> pd.concat([df,df["A"]], axis=1)
                   A         B         C         D         A
2018-01-31 -0.036007  2.425324 -0.160817       NaN -0.036007
2018-02-28  1.458603 -0.687165  1.458743  0.234733  1.458603
2018-03-31 -0.860041       NaN -0.138422       NaN -0.860041
2018-04-30 -1.082399  0.274795  0.935195  0.939475 -1.082399
2018-05-31 -1.386264  0.845519  2.252664  0.216155 -1.386264
2018-06-30 -0.926516  0.049159  0.094237  1.341607 -0.926516
```

## 7 Reading and Writing Files

PANDAS offers two functions to read input files and load their content into data structures. The first one is `read_table()`, which allows to read a general delimited file into a `DataFrame`. The second one is `read_csv()`, allowing to read a *Comma Separated File* (CSV) into a `DataFrame`.

A CSV file is a plaintext file containing letters and numbers only, and structuring the data it contains in a table form. In CSV files, columns are separated by commas.

By default, `read_table()` uses the character '\t' as separator, while `read_csv()` uses ','. This default behavior can be modified with the *sep* or *delimiter* arguments. If the separator is a whitespace (i.e. ' ' or '\t', for example), you can directly set the argument *delim_whitespace* to *True*, without using *sep* or *delimiter*.

If you need to write structures to CSV files, you can the method `to_csv()`. The column separator can be modified using the argument *sep*. Note that PANDAS does not implement a writing method similar to `read_table()`. You must then use `to_csv()`, and change the separator and/or the file extension. Another solution is to use the NUMPY function `savetxt()`. In this case, you must extract the data from the structure before writing the file, as NUMPY does not recognize PANDAS' structures.

When reading a file, the first line will be considered as the column labels, by default.

```
# Read from a CSV file
>>> df = pd.read_csv("foo.csv")

# Write to a CSV file, specifying \t as delimiter
>>> df.to_csv("foo.csv", sep='\t')
```