Scientific Programming

Lecture A07 – Pandas

Andrea Passerini

Università degli Studi di Trento

2019/10/22

Acknowledgments: Alberto Montresor, Stefano Teso, Pandas Documentation

Table of contents

1. Introduction
2. Series
3. DataFrames

What is Pandas?

Pandas

A freely available library for loading, manipulating, and visualizing sequential and tabular data, such as time series or micro-arrays.

Features

- Loading and saving with "standard" tabular file formats:
  - CSV (Comma-separated Values)
  - TSV (Tab-separated Values)
  - Excel files
  - Database formats, etc.
- Flexible indexing and aggregation of series and tables
- Efficient numerical/statistical operations (e.g. broadcasting)
- Pretty, straightforward visualization

Some links

Official Pandas website

http://pandas.pydata.org/

Official documentation

http://pandas.pydata.org/pandas-docs/stable/dsintro.html

Source code

https://github.com/pandas-dev/pandas/

A short demonstration – Iris Dataset

```
SepalLength,SepalWidth,PetalLength,PetalWidth,Name
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
...
5.0,3.3,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
...
5.7,2.8,4.1,1.3,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
...
```

https://drive.google.com/open?id=0B0wILN942aEVYTVBekRHLTN0N3c
https://en.wikipedia.org/wiki/Iris_flower_data_set

A short demonstration – Iris Dataset

In an effort to understand the dataset, we would like to visualize the relation between the four properties for the case of Iris virginica.

- Load the dataset by parsing all the rows in the file
- Keep only the rows pertaining to Iris virginica
- Compute statistics on the values of the rows, making sure to convert from strings to float‘s as required
- Actually draw the plots by using a specialized plotting library.

A short demonstration – Iris Dataset

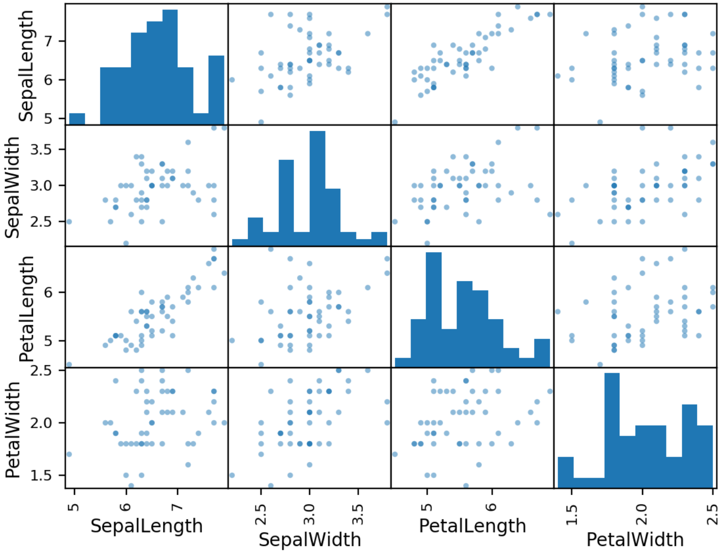
```
import pandas as pd
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt

df = pd.read_csv("iris.csv")

scatter_matrix(df[df.Name == "Iris-virginica"])

plt.show()
```

A short demonstration – Iris Dataset



Introduction to Pandas

Pandas provides a couple of very useful datatypes:

- `Series` represents 1D data, like time series, calendars, the output of one-variable functions, etc.
- `DataFrame` represents 2D data, like a column-separated-values (CSV) file, a microarray, a database table, a matrix, etc.

Each column of a `DataFrame` is a `Series`.

- That's why we will see how the `Series` data type works first.
- Most of what we will say about `Series` also applies to `DataFrame`s.

Table of contents

1. Introduction
2. Series
3. DataFrames

Pandas: `Series`

Series

A `Series` is a one-dimensional array with a labeled axis, that can hold arbitrary objects.

The axis is called the index, and can be used to access the elements; it is very flexible, and not necessarily numerical.

It works partially like a `list` and partially like a `dict`.

Creating a Series (1)

It is possible to specify just the series data, associating an implicit numeric index.

```
import pandas as pd
s = pd.Series(["a", "b", "c"])
print(s)
```

```
0    a
1    b
2    c
dtype: object
```

Creating a Series (2)

It is possible to specify both the series data and the explicit index, separately:

```
import pandas as pd
s = pd.Series(["a", "b", "c"], index=[2, 5, 8])
print(s)
```

```
2    a
5    b
8    c
dtype: object
```

Creating a Series (3)

It is possible to specify both the series data and the index, as a single dictionary:

```
import pandas as pd
s = pd.Series({"a": "A", "b": "B", "c": "C"})
print(s)
```

```
a    A
b    B
c    C
dtype: object
```

Creating a Series (4)

If given a single scalar (e.g. an integer), the series constructor will replicate it for all indices (that need to be specified)

```
import pandas as pd
s = pd.Series(3, index=range(5))
print(s)
```

```
0    3
1    3
2    3
3    3
4    3
dtype: int64
```

Accessing a Series

Let's create a Series representing the hours of sleep we had the chance to get each day of the past week. We may now access it through either the position (as a list) or the index (as a dict)

```
import pandas as pd
days = ["mon", "tue", "wed", "thu", "fri"]
sleephours = [6, 2, 8, 5, 9]
s = pd.Series(sleephours, index=days)
print(s["mon"])
s["tue"]=3
print(s[1])
```

```
6
3
```

Accessing a Series

- If a label is not contained, an exception is raised.
- Using the get method, a missing label will return `None` or specified default

```
import pandas as pd
days =        ["mon", "tue", "wed", "thu", "fri"]
sleephours = [6,      2,     8,     5,     9]
s = pd.Series(sleephours, index=days)

print(s["sat"])
print(s.get('sat'))
```

```
KeyError: 'sat'
None
```

Slicing a Series

We can also slice the positions, like we would do with a list. Note that both the data and the index are extracted correctly. It also works with labels.

```
print(s[-3:])
print(s["tue":"thu"])
```

```
wed    8
thu    5
fri    9
dtype: int64
tue    2
wed    8
thu    5
dtype: int64
```

Head and tail

The first and last `n` elements can be extracted also using `head()` and `tail()`.

```
print(s.head(2))
print(s.tail(3))
```

```
mon    6
tue    2
dtype: int64

wed    8
thu    5
fri    9
dtype: int64
```

List of indexes

You can also explicitly pass a list of positions. Tuples do not work, because they are interpreted as potential indexes.

```
print(s[[0, 1, 2]])
print(s[["mon", "wed", "fri"]])
```

```
mon     6
tue     2
wed     8
dtype: int64

mon     6
wed     8
fri     9
dtype: int64
```

Operator broadcasting

The `Series` class automatically broadcasts arithmetical operations by a scalar to all of the elements.

```
print(s)

mon      6
tue      2
wed      8
thu      5
fri      9
dtype: int64
```

```
print(s+1)

mon       7
tue       3
wed       9
thu       6
fri      10
dtype: int64
```

```
print(s*2)

mon      12
tue       4
wed      16
thu      10
fri      18
dtype: int64
```

Note

The concept of operator broadcasting was taken from the `numpy` library, and is one of the key features for writing efficient, clean numerical code in Python.

In a way, it is a “generalized” version of scalar products (from linear algebra).

The rules governing how broadcasting is applied can be pretty complex (and confusing). For the moment, we will cover constant broadcasting only.

Masking and filtering

- Besides numerical operators, we can apply boolean conditions. The result is called a mask.
- Masks can be used to filter the elements of a `Series` according to a given condition.

```
print(s)

mon     6
tue     2
wed     8
thu     5
fri     9
dtype: int64
```

```
print(s>=6)

mon      True
tue     False
wed      True
thu     False
fri      True
dtype: bool
```

```
print(s[s>=6])

mon      6
wed      8
fri      9
dtype: int64
```

Automatic label assignments

Operations between multiple time series are automatically aligned by label, meaning that elements with the same label are matched prior to carrying out the operation.

```
print(s[1:])

tue     2
wed     8
thu     5
fri     9
dtype: int64
```

```
print(s[:-1])

mon     6
tue     2
wed     8
thu     5
dtype: int64
```

```
print(s[1:]+s[:-1])

fri      NaN
mon      NaN
thu     10.0
tue      4.0
wed     16.0
dtype: float64
```

Not-a-Number (`NaN`)

The index of the resulting `Series` is the union of the indices of the operands. What happens depend on whether a given label appears in both input Series or not:

- For common labels (in our case `"tue"`, `"wed"`, `"thu"`), the output `Series` contains the sum of the aligned elements.
- For labels appearing in only one of the operands (`"mon"` and `"fri"`), the result is a `NaN`, i.e. not-a-number.

`NaN` is just a symbolic constant that specifies that the object is a number-like entity with an invalid or undefined value.

Dealing with missing values

There are different strategies for dealing with nan's. There is no "best" strategy: you have to pick one depending on the problem you are trying to solve.

```
t = s[1:]+s[:-1]
print(t)

fri      NaN
mon      NaN
thu     10.0
tue      4.0
wed     16.0
dtype: float64
```

```
nt = t.dropna()
print(nt)

thu     12.0
tue      6.0
wed     18.0
dtype: float64
```

```
zt = t.fillna(0.0)
print(zt)

fri      0.0
mon      0.0
thu     12.0
tue      6.0
wed     18.0
dtype: float64
```

Automatic label assignments

Through the method add, it is possible to assign a fill value to the missing entries of the series to be added, in order to get a real sum.

```
print(s[1:])

tue     2
wed     8
thu     5
fri     9
dtype: int64
```

```
print(s[:-1])

mon     6
tue     2
wed     8
thu     5
dtype: int64
```

```
print(s[1:].add(s[:-1],
  fill_value=0))

fri      9.0
mon      6.0
thu     10.0
tue      4.0
wed     16.0
dtype: float64
```

Computing statistics

```python
print(s)
```

```
mon      6
tue      2
wed      8
thu      5
fri      9
dtype: int64
```

```python
print(s.sum())
print(s.prod())
print(s.max())
print(s.argmax())
print(s.mean())
print(s.var())
print(s.std())
print(s.median())
```

```
30
4320
9
fri
6.0
7.5
2.7386127875258306
6.0
```

```python
print(s.cumsum())
```

```
mon     6
tue     8
wed    16
thu    21
fri    30
dtype: int64
```

Computing statistics

```
print(s)

mon     6
tue     2
wed     8
thu     5
fri     9
dtype: int64
```

```
print(s.quantile(0.5))
```

```
6.0
```

```
print(s.quantile(
  [0.25, 0.5, 0.75]
))
```

```
0.25    5.0
0.50    6.0
0.75    8.0
dtype: float64
```

```
print(s[s >=
  s.quantile(0.5)
])
```

```
mon    6
wed    8
fri    9
dtype: int64
```

Computing statistics

```
print(s)

mon     6
tue     2
wed     8
thu     5
fri     9
dtype: int64
```

```
# Pearson corr.
print(s.corr(s))                  1.0

# Spearman corr.
print(s.corr(s,                   1.0
  method="spearman"
)

# Autocorrelation
# with time lag
print(s.autocorr(lag=0))    1.0
print(s.autocorr(lag=1))    -0.548128127763
print(s.autocorr(lag=2))    0.995870594886
```

Series: Conclusion

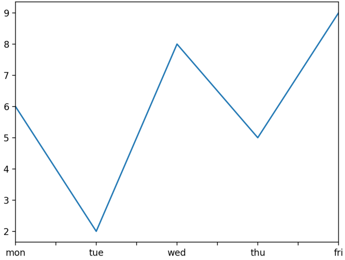
A quick way to get several useful statistics is to use `s.describe()`. Anyway, the list of statistical methods associated with `Series` is larger than this.

```
print(s.describe())
```

```
count     5.000000
mean      6.000000
std       2.738613
min       2.000000
25%       5.000000
50%       6.000000
75%       8.000000
max       9.000000
dtype: float64
```

Series: Plotting

```
import pandas as pd
import matplotlib.pyplot as plt
sleephours = [6, 2, 8, 5, 9]
days = ["mon", "tue", "wed", "thu", "fri"]
s = pd.Series(sleephours, index=days)
s.plot()
plt.show()
```




```
import pandas as pd
import matplotlib.pyplot as plt
sleephours = [6, 2, 8, 5, 9]
days = ["mon", "tue", "wed", "thu", "fri"]
s = pd.Series(sleephours, index=days)
s.plot(kind="bar")
plt.show()
```

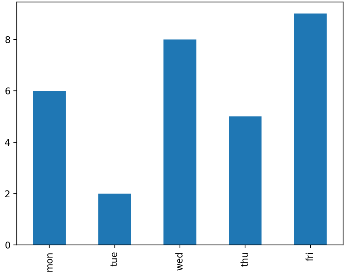


Table of contents

1. Introduction
2. Series
3. DataFrames

DataFrame

Pandas `DataFrame` is the 2D analogue of a `Series`: it is essentially a table of heterogeneous objects.

A `DataFrame` holds three major attributes:

- the index, which holds the labels of the rows
- the columns, which holds the labels of the columns
- the shape, which describes the dimension of the table

When you extract a column from a `DataFrame` you get a proper `Series`, and you can operate on it using all the tools presented in the previous section.

Further, most (not all) of the operations that you can do on a `Series`, you can also do on an entire `DataFrame`.

Creating a DataFrame from a dictionary of Series

```
import pandas as pd
d = { "name": pd.Series(["bobby", "ronald", "ronald", "ronald"]),
  "surname": pd.Series(["fisher", "fisher", "reagan", "mcdonald"]) }
df = pd.DataFrame(d)
print(df)
print(df.columns)
print(df.index)
print(df.shape)
```

```
     name   surname
0   bobby    fisher
1  ronald    fisher
2  ronald    reagan
3  ronald  mcdonald
Index(['name', 'surname'], dtype='object')
RangeIndex(start=0, stop=4, step=1)
(4, 2)
```

- The keys of the dictionary became the columns of the dataframe
- The index of the various `Series` became the index of the dataframe.

Creating a DataFrame from a dictionary of Series

```
import pandas as pd
d = { "x": pd.Series([0, 0], index=["a", "b"]),
      "y": pd.Series([0, 0], index=["b", "c"])}
df = pd.DataFrame(d)
print(df)
print(df.columns)
print(df.index)
print(df.shape)
```

- If the index of the input `Series` do not match, since label alignment applies, the missing values are treated as `NaN`‘s.

```
     x    y
a  0.0  NaN
b  0.0  0.0
c  NaN  0.0
Index(['x', 'y'], dtype='object')
Index(['a', 'b', 'c'], dtype='object')
(3,2)
```

Creating a DataFrame from a dictionary of lists

```
import pandas as pd
d = { "column1": [1., 2., 6., -1.],
      "column2": [0., 1., -2., 4.] }
df = pd.DataFrame(d)
print(df)
print(df.columns)
print(df.index)
print(df.shape)
```

- The columns are taken from the keys
- The index is set to the default one

```
   column1  column2
0      1.0      0.0
1      2.0      1.0
2      6.0     -2.0
3     -1.0      4.0
Index(['column1', 'column2'], dtype='object')
RangeIndex(start=0, stop=4, step=1)
(4,2)
```

Creating a `DataFrame` from a dictionary of `lists`, with index

```
import pandas as pd
d = { "column1": [1., 2., 6., -1.],
      "column2": [0., 1., -2., 4.] }
df = pd.DataFrame(d, index=["a", "b", "c", "d"])
print(df)
print(df.columns)
print(df.index)
print(df.shape)
```

- A custom index can be specified the usual way

```
   column1  column2
a      1.0      0.0
b      2.0      1.0
c      6.0     -2.0
d     -1.0      4.0
Index(['column1', 'column2'], dtype='object')
RangeIndex(start=0, stop=4, step=1)
(4,2)
```

Creating a DataFrame from a list of dictionaries

```
import pandas as pd
d = [ {"a": 1, "b": 2},
      {"a": 2, "c": 3},
]
df = pd.DataFrame(d)
print(df)
print(df.columns)
print(df.index)
print(df.shape)
```

- The columns are taken from the keys of the dictionaries
- The index is the default one.
- Since not all common keys appear in all input dictionaries, missing values (i.e. `NaN`'s) are automatically added.

```
   a    b    c
0  1  2.0  NaN
1  2  NaN  3.0
Index(['a', 'b', 'c'], dtype='object')
RangeIndex(start=0, stop=2, step=1)
(2,3)
```

Loading a CSV file

```python
import pandas as pd
df = pd.read_csv("iris.csv")
print(df.columns)
print(df.index)
print(df.shape)
```

```
Index(['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Name'],
  dtype='object')
RangeIndex(start=0, stop=150, step=1)
(150, 5)
     SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0            5.1         3.5          1.4         0.2  Iris-setosa
1            4.9         3.0          1.4         0.2  Iris-setosa
2            4.7         3.2          1.3         0.2  Iris-setosa
3            4.6         3.1          1.5         0.2  Iris-setosa
4            5.0         3.6          1.4         0.2  Iris-setosa
5            5.4         3.9          1.7         0.4  Iris-setosa
...
```

help(pd.read_csv)

```
Help on function read_csv in module pandas.io.parsers:

read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer',
names=None, index_col=None, usecols=None, squeeze=False, prefix=None,
mangle_dupe_cols=True, dtype=None, engine=None, converters=None,
true_values=None, false_values=None, skipinitialspace=False,
skiprows=None, nrows=None, na_values=None, keep_default_na=True,
na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False,
infer_datetime_format=False, keep_date_col=False, date_parser=None,
dayfirst=False, iterator=False, chunksize=None, compression='infer',
thousands=None, decimal=b'.', lineterminator=None, quotechar='"',
quoting=0, escapechar=None, comment=None, encoding=None, dialect=None,
tupleize_cols=False, error_bad_lines=True, warn_bad_lines=True,
skipfooter=0, skip_footer=0, doublequote=True, delim_whitespace=False,
as_recarray=False, compact_ints=False, use_unsigned=False,
low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)

Read CSV (comma-separated) file into DataFrame
```

Loading a malformed Tab-Separated File

```
https://drive.google.com/open?id=0B0wILN942aEVeWdScy1wQnA3LT
```

It describes a mapping from UniProt protein IDs (i.e. sequences) to hits in the Protein Data Bank (i.e. structures). The TSV file looks like this:

```
# 2014/07/08 - 14:59
SP_PRIMARY  PDB
A0A011      3vk5;3vka;3vkb;3vkc;3vkd
A0A2Y1      2jrd
A0A585      4mnq
A0A5B4      2ij0
A0A5B9      2bnq;2eyr;2eys;2eyt;3kxf;3o6f;3o8x;3o9w;3qux;3t0e;3
A0A5E3      1hq4;1ob1
A0AEF5      4iu2;4iu3
A0AEF6      4iu2;4iu3
A0A007      2ib5
```

Loading a malformed Tab-Separated File

We can use the `sep` (separator) argument of the `read_csv()` method to take care of the TABs.

```
df = pd.read_csv("uniprot_pdb.tsv", sep="\t")
print(df.shape)
print(df.columns)
print(df.head(3))
```

```
(33636, 1)
Index(['# 2014/07/08 - 14:59'], dtype='object')
                # 2014/07/08 - 14:59
SP_PRIMARY                       PDB
A0A011      3vk5;3vka;3vkb;3vkc;3vkd
A0A2Y1                          2jrd
```

Problem: the first line contains only one column, so Pandas think that there is only one column

Loading a malformed Tab-Separated File

Argument skiprows is used to skip the first line, which is a comment.

```
df = pd.read_csv("uniprot_pdb.tsv", sep="\t", skiprows=1)
print(df.shape)
print(df.columns)
print(df.head(3))
```

```
(33636, 2)
Index(['SP_PRIMARY', 'PDB'], dtype='object')
  SP_PRIMARY                                            PDB
0     A0A011                       3vk5;3vka;3vkb;3vkc;3vkd
1     A0A2Y1                                           2jrd
2     A0A585                                           4mnq
```

Loading a malformed Tab-Separated File

Argument `comment` is used to skip all the lines that start with #

```
df = pd.read_csv("uniprot_pdb.tsv", sep="\t", comment="#")
print(df.shape)
print(df.columns)
print(df.head(3))
```

```
(33636, 2)
Index(['SP_PRIMARY', 'PDB'], dtype='object')
  SP_PRIMARY                                          PDB
0     A0A011                     3vk5;3vka;3vkb;3vkc;3vkd
1     A0A2Y1                                         2jrd
2     A0A585                                         4mnq
```

Extracting rows and columns

Operation	Syntax	Result
Select column	`df[col]`	`Series`
Select multiple columns	`df[[col1,col2]]`	`DataFrame`
Select row by label	`df.loc[label]`	`Series`
Select row by integer location	`df.iloc[loc]`	`Series`
Slice rows	`df[5:10]`	`DataFrame`
Select rows by boolean vector	`df[bool_vec]`	`DataFrame`

Extracting a subset of the Iris Dataset

For simplicity, in the following examples we will use a random sample taken from the iris dataset, computed like this

```
import numpy as np
import pandas as pd
np.random.seed(0)
df = pd.read_csv("iris.csv")
small = df.iloc[np.random.permutation(df.shape[0])].head()
print(small.shape)
print(small)
```

Brief explanation: here we use `numpy.random.permutation()` to generate a random permutation of the indices from `0` to `df.shape[0]`, i.e. the number of rows in the Iris dataset; then we use this permutation as row indices to permute all the rows in `df`; finally, we take the first 5 rows of the permuted `df` using the `head()` method.

Extracting a subset of the Iris Dataset

```
(5, 5)
     SepalLength  SepalWidth  PetalLength  PetalWidth            Na
114          5.8         2.8          5.1         2.4   Iris-virgin
62           6.0         2.2          4.0         1.0  Iris-versico
33           5.5         4.2          1.4         0.2      Iris-set
107          7.3         2.9          6.3         1.8   Iris-virgin
7            5.0         3.4          1.5         0.2      Iris-set
```

Extracting a column

It is possible to access the columns of `small` with the `[]` notation. The result is a `Series`.

If the name of the column is compatible with the Python conventions for variable names, you can also treat columns as if they were actual attributes of the dataframes.

```
print(small["Name"])        OR        print(small.Name)
```

```
114     Iris-virginica
62     Iris-versicolor
33         Iris-setosa
107     Iris-virginica
7          Iris-setosa
Name: Name, dtype: object
```

Extracting multiple columns

It is possible to extract multiple columns in one go, by specifying a list of columns; the result is a `DataFrame`

```
print(small[["SepalLength", "PetalLength"]])
```

```
     SepalLength  PetalLength
114          5.8          5.1
62           6.0          4.0
33           5.5          1.4
107          7.3          6.3
7            5.0          1.5
```

Extracting a row

To extract a row, it is possible to use the `loc` and `iloc` attributes by specifying a label or a position, respectively.
The result is a `Series`

```
print(small.loc[114])        OR        print(small.iloc[0])
```

```
SepalLength                5.8
SepalWidth                 2.8
PetalLength                5.1
PetalWidth                 2.4
Name            Iris-virginica
Name: 114, dtype: object
```

Extracting multiple rows

To extract multiple rows, it is possible to use the `loc` and `iloc` attributes by specifying a list of labels or positions.
The result is a `Dataframe`

```
print(small.loc[[114,62,33]]) OR print(small.iloc[[0,1,2]])
```

```
     SepalLength  SepalWidth  PetalLength  PetalWidth             Name
114          5.8         2.8          5.1         2.4   Iris-virginica
62           6.0         2.2          4.0         1.0  Iris-versicolor
33           5.5         4.2          1.4         0.2      Iris-setosa
```

Broadcasting

Broadcasting is applied automatically to all rows, or to the entire table.

```
print(small["SepalLength"] + small["SepalWidth"])
print(small+small)
```

```
114     8.6
62      8.2
33      9.7
107    10.2
7       8.4
dtype: float64
```

```
     SepalLength  SepalWidth  PetalLength  PetalWidth
114         11.6         5.6         10.2         4.8   Iris-virginicaIris-
62          12.0         4.4          8.0         2.0 Iris-versicolorIris-v
33          11.0         8.4          2.8         0.4         Iris-setosaIr
107         14.6         5.8         12.6         3.6   Iris-virginicaIris-
7           10.0         6.8          3.0         0.4         Iris-setosaIr
```

Masking

Masking works as well.

```
print(small["PetalLength"][small.PetalLength > 5])
print(small["Name"][small.Name == "Iris-virginica"])
print(small[["Name","PetalLength","SepalLength"][
                      small.Name == "Iris-virginica"])
```

```
114    5.1
107    6.3
Name: PetalLength, dtype: float64

114    Iris-virginica
107    Iris-virginica
Name: Name, dtype: object

               Name  PetalLength  SepalLength
114  Iris-virginica          5.1          5.8
107  Iris-virginica          6.3          7.3
```

Statistics

Statistics can be computed on rows, columns, or the whole table.

```
print(small.loc[114][:-1].mean()) # Exclude last column, Name
print(small.PetalLength.mean())
print(small.mean())
```

```
4.025

3.66

SepalLength    5.92
SepalWidth     3.10
PetalLength    3.66
PetalWidth     1.12
dtype: float64
```

All together!

```
print(small[["Name", "PetalLength", "SepalLength"]][
  small.PetalLength > small.PetalLength.mean()])
```

```
                Name  PetalLength  SepalLength
114   Iris-virginica          5.1          5.8
62   Iris-versicolor          4.0          6.0
107   Iris-virginica          6.3          7.3
```

Merge

Merging different dataframes is performed using the `merge()` function.

Merging means that, given two tables with a common column name, first the rows with the same column value are matched; then a new table is created by concatenating the matching rows.

```
sequences = pd.DataFrame({
    "id":  ["Q99697", "O18400", "P78337", "Q9W5Z2"],
    "seq": ["METNCR", "MDRSSA", "MDAFKG", "MTSMKD"],
})

names = pd.DataFrame({
    "id":   ["Q99697", "O18400", "P78337", "P59583"],
    "name": ["PITX2_HUMAN", "PITX_DROME",
             "PITX1_HUMAN", "WRK32_ARATH"],
})
```

Merge - Inner

```
print(sequences)
print(names)
print(pd.merge(sequences, names, on="id", how="inner"))
```

```
       id     seq
0  Q99697  METNCR
1  O18400  MDRSSA
2  P78337  MDAFKG
3  Q9W5Z2  MTSMKD
```

```
       id         name
0  Q99697  PITX2_HUMAN
1  O18400   PITX_DROME
2  P78337  PITX1_HUMAN
3  P59583  WRK32_ARATH
```

```
       id     seq         name
0  Q99697  METNCR  PITX2_HUMAN
1  O18400  MDRSSA   PITX_DROME
2  P78337  MDAFKG  PITX1_HUMAN
```

Mismatched ids are dropped

Merge - Left

```
print(sequences)
print(names)
print(pd.merge(sequences, names, on="id", how="left"))
```

```
       id     seq
0  Q99697  METNCR
1  O18400  MDRSSA
2  P78337  MDAFKG
3  Q9W5Z2  MTSMKD
```

```
       id         name
0  Q99697  PITX2_HUMAN
1  O18400   PITX_DROME
2  P78337  PITX1_HUMAN
3  P59583  WRK32_ARATH
```

```
       id     seq         name
0  Q99697  METNCR  PITX2_HUMAN
1  O18400  MDRSSA   PITX_DROME
2  P78337  MDAFKG  PITX1_HUMAN
3  Q9W5Z2  MTSMKD          NaN
```

The ids are taken from the left table

Merge - Right

```
print(sequences)
print(names)
print(pd.merge(sequences, names, on="id", how="right"))
```

```
       id     seq
0  Q99697  METNCR
1  O18400  MDRSSA
2  P78337  MDAFKG
3  Q9W5Z2  MTSMKD
```

```
       id         name
0  Q99697  PITX2_HUMAN
1  O18400   PITX_DROME
2  P78337  PITX1_HUMAN
3  P59583  WRK32_ARATH
```

```
       id     seq         name
0  Q99697  METNCR  PITX2_HUMAN
1  O18400  MDRSSA   PITX_DROME
2  P78337  MDAFKG  PITX1_HUMAN
3  P59583     NaN  WRK32_ARATH
```

The ids are taken from the right table

Merge - Outer

```
print(sequences)
print(names)
print(pd.merge(sequences, names, on="id", how="outer"))
```

```
       id     seq
0  Q99697  METNCR
1  O18400  MDRSSA
2  P78337  MDAFKG
3  Q9W5Z2  MTSMKD
```

```
       id         name
0  Q99697  PITX2_HUMAN
1  O18400   PITX_DROME
2  P78337  PITX1_HUMAN
3  P59583  WRK32_ARATH
```

```
       id     seq         name
0  Q99697  METNCR  PITX2_HUMAN
1  O18400  MDRSSA   PITX_DROME
2  P78337  MDAFKG  PITX1_HUMAN
3  Q9W5Z2  MTSMKD          NaN
4  P59583     NaN  WRK32_ARATH
```

All ids are retained

Grouping Tables

The `groupby` method is essential for efficiently performing operations on groups of rows.

Given the Iris dataset, we want to compute the average of the four columns for each of the three different Iris species.

The result should be a dataframe with 3 species (rows) by 4 columns (petal/sepal length/width).

Grouping Tables

```
import pandas as pd
iris = pd.read_csv("iris.csv")
print(iris.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
SepalLength    150 non-null float64
SepalWidth     150 non-null float64
PetalLength    150 non-null float64
PetalWidth     150 non-null float64
Name           150 non-null object
dtypes: float64(4), object(1)
memory usage: 5.9+ KB
```

Grouping Tables

Iterating over the grouped variable returns (value-of-Name, DataFrame) tuples:

- The 1st item is the value of the column `Name` shared by the group
- The 2nd item is a `DataFrame` including only the rows in that group.

```
grouped = iris.groupby(iris.Name)
for group in grouped:
    print(group[0], group[1].shape)
```

```
Iris-setosa (50, 5)
Iris-versicolor (50, 5)
Iris-virginica (50, 5)
```

Grouping Tables

It is possible to apply some transformation (e.g. `mean()`) to the individual groups automatically, using the `aggregate()` method directly on the grouped variable. The result of `aggregate()` is a dataframe.

```
iris_mean_by_name = grouped.aggregate(pd.DataFrame.mean)
print(iris_mean_by_name)
```

```
                 SepalLength  SepalWidth  PetalLength  PetalWidth
Name
Iris-setosa            5.006       3.418        1.464       0.244
Iris-versicolor        5.936       2.770        4.260       1.326
Iris-virginica         6.588       2.974        5.552       2.026
```