

# Python Basic Course

*Part IV*

Stefano Alberto Russo

# Outline

- Part I: introduction and basics

- What is Python
- Tools and “hello world”
- Basic syntax and data types
  - assignments, types and operators
  - conditional blocks and loops

- Part II: architecture

- Functions
- Scope
- Built-ins
- Modules

- Part IV: manipulating data

- List operations
- String operations
- Dealing bad data
- Reading and writing files

- Part VI: Pandas

- Series and Dataframes
- Common operations
- How to read documentation

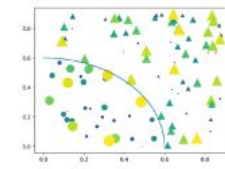
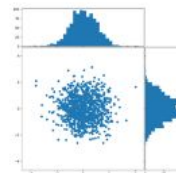
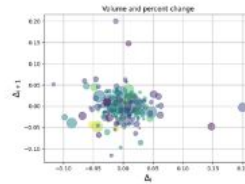
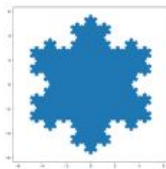
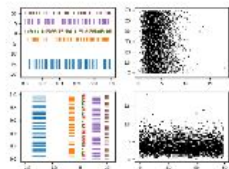
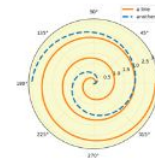
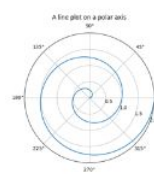
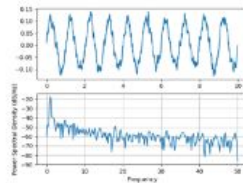
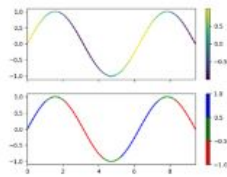
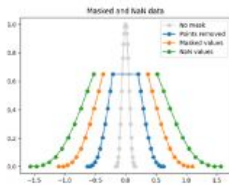
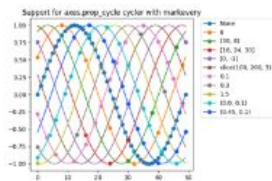
# Pandas

## → *What is Pandas?*

- Pandas is a “fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language”
- Offers data structures and operations for manipulating numerical tables in form of arrays and matrices, and time series to some extent.
- Pandas *does not* marry entirely the Python philosophy: often requires working with indexes to iterate over data structures and adopting an “old-fashioned” mindset.
- The name is derived from the term “panel data”, an econometrics term for data sets that include observations over multiple time periods for the same individuals.

# Pandas

→ *What is Pandas?*



# Pandas

→ *How to install Pandas?*

- Being an extra Python library, it needs to be installed.
- The Python Package Manager can be used for this in nearly all environments:

```
$ pip install pandas
```

- In Repl.it, it is automatically installed, so you don't have to worry about it.

# Pandas

→ *How to use Pandas?*

- As any other library, Pandas needs to be imported before you can use it
- You import libraries exactly as you import modules:

```
import pandas
```

- You will usually see it imported in a renamed way, for brevity when using it

```
import pandas as pd
```

# Pandas

→ *Other libraries involved*

- Numpy (Numerical Python) is another very common library used together with Pandas:

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

- Matplotlib is instead a library for plotting, and in particular the pyplot module is very commonly used:

```
import matplotlib.pyplot as plt
```

# Pandas

## → *The Jupyter Notebooks*

- Pandas has a strong interactive component and for interactive analysis gives it best when used with the Jupyter Notebooks
- These are computational graphical environments which wrap a Python interpreter
- Several services derived from this approach, as Google Colab or Kaggle Notebooks.
- Installing and using the Jupyter engine in your environment it is not covered here, but just for reference:

```
$ pip install notebook
```

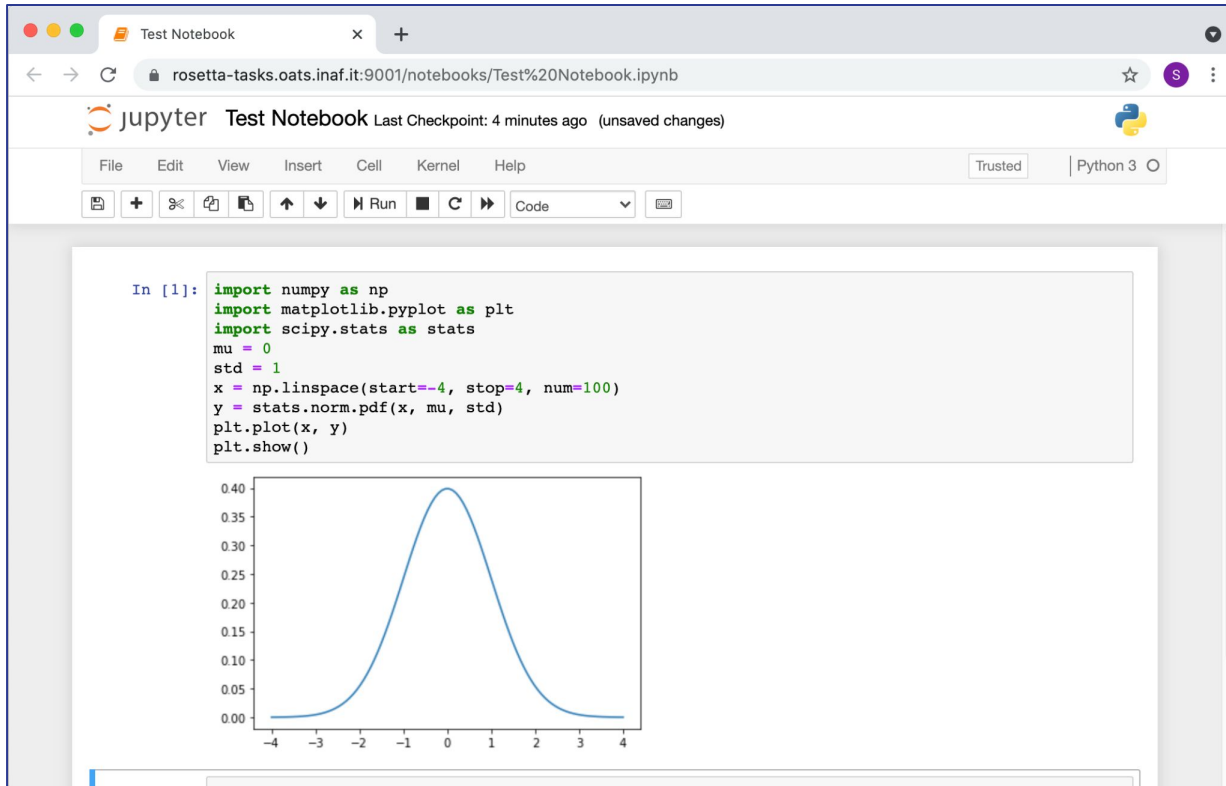
```
$ jupyter notebook
```

..and then open your browser on localhost:8888



# Pandas

→ *The Jupyter Notebooks*



# Pandas

## → *Series*

- Pandas Series are one of the most basic data types. You can think of them as Python lists, but provide much more features.

```
series = pd.Series([4, 5, 6])  
print(series[0])
```

4

# Pandas

## → *Series*

- Series have an index to speed up data access. By defaults, it is just composed by the positions of the elements

```
series = pd.Series([4,5,6])  
print(series)
```

```
0    4  
1    5  
2    6  
dtype: int64
```

# Pandas

## → Series

- Series have an index to speed up data access. By defaults, it is just composed by the positions of the elements

```
series = pd.Series([4,5,6])  
print(series)
```

Index

```
0    4  
1    5  
2    6  
dtype: int64
```

# Pandas

## → *Series*

- However, other types of indexes are possible, for example based on letters, or dates and time. They are more complex to deal with.

```
series = pd.Series([4,5,6])  
series.index = ['a','b','c']
```

```
a    4  
b    5  
c    6  
dtype: int64
```

# Pandas

## → *Series*

- However, other types of indexes are possible, for example based on letters, or dates and time. They are more complex to deal with.

```
series = pd.Series([4,5,6])  
series.index = ['a','b','c']  
print(series[0])
```

4

# Pandas

## → *Series*

- However, other types of indexes are possible, for example based on letters, or dates and time. They are more complex to deal with.

```
series = pd.Series([4,5,6])  
series.index = ['a','b','c']  
print(series['a'])
```

4

# Pandas

## → *Series*

- However, other types of indexes are possible, for example based on letters, or dates and time. They are more complex to deal with.

```
series = pd.Series([4,5,6])  
series.index = ['a','b','c']  
print(series.iloc[0])
```

4



# Pandas

## → *Series*

- However, other types of indexes are possible, for example based on letters, or dates and time. They are more complex to deal with.

```
series = pd.Series([4,5,6])  
series.index = ['a','b','c']  
print(series.loc['a'])
```

4

# Pandas

## → *Series*

- However, other types of indexes are possible, for example based on letters, or dates and time. They are more complex to deal with.

```
series = pd.Series([4,5,6])  
series.index = ['a','b','c']  
print(series.loc['a'])
```

4

# Pandas

## → *Series*

- Both Series and their indexes supports being iterated on, and allow to be more pythonic in some contexts:

```
series = pd.Series([4,5,6])  
for item in series:  
    print(item)
```

```
4  
5  
6
```

# Pandas

## → *Series*

- Both Series and their indexes supports being iterated on, and allow to be more pythonic in some contexts:

```
series = pd.Series([4,5,6])
for index_item in series.index:
    print(index_item)
```

```
0
1
2
```

# Pandas

## → *Series*

- Several functions are ready to be applied to the series, unlike the Python lists. Mean, min, max etc. are just some examples of them.

```
series = pd.Series([4, 5, 6])  
print(series.mean())
```

```
5.0
```

# Pandas

## → *Series*

- Several functions are ready to be applied to the series, unlike the Python lists. Mean, min, max etc. are just some examples of them.

```
series = pd.Series([4, 5, 6])  
print(series.max())
```

```
6.0
```

# Pandas

## → *DataFrames*

- DataFrames are basically matrices. They support multiple axes, indexes, and labels for columns.

```
df = pd.DataFrame([[4, 40], [5, 50], [6, 60]])  
print(df)
```

```
   0  1  
0  4  40  
1  5  50  
2  6  60
```

# Pandas

## → DataFrames

- DataFrames are basically matrices. They support multiple axes, indexes, and labels for columns.

```
df = pd.DataFrame([[4, 40], [5, 50], [6, 60]])  
print(df)
```

**Column labels**

	0	1
0	4	40
1	5	50
2	6	60

**Index**



# Pandas

## → *DataFrames*

- If accessing them by “position”, they return a column which is returned as Series which “inherits” the index

```
df = pd.DataFrame([[4, 40], [5, 50], [6, 60]])  
print(df[1])
```

```
0    40  
1    50  
2    60  
Name: 1, dtype: int64
```

# Pandas

## → *DataFrames*

- If accessing them by “position”, they return a column which is returned as as Series which “inherits” the index

```
df = pd.DataFrame([[4, 40], [5, 50], [6, 60]])  
type(df[1])
```

```
pandas.core.series.Series
```

# Pandas

## → *DataFrames*

- Data frames supports changing not only the index but also the column labels:

```
df = pd.DataFrame([[4, 40], [5, 50], [6, 60]])  
df.index = ['a', 'b', 'c']  
df.columns = ['Rome', 'Venice']  
print(df)
```

	Rome	Venice
a	4	40
b	5	50
c	6	60

# Pandas

## → DataFrames

- DataFrames can also be created directly from Python dictionaries, but remember that you will not have any order guaranteed in the columns!

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
  
print(df)
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

# Pandas

## → DataFrames

- DataFrames can also be created directly from Python dictionaries, but remember that you will not have any order guaranteed in the columns!

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
  
print(df)
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

	Rome	Venice
0	4	40
1	5	50
2	6	60



# Pandas

## → DataFrames

- DataFrames can also be created directly from Python dictionaries, but remember that you will not have any order guaranteed in the columns!

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
  
print(df)
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

# Pandas

## → DataFrames

- At this point you can access the columns using their label in the square brackets notation. Keep in mind that for the Series, this was instead accessing the “rows”.

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
print(df['Venice'])
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

```
0    40  
1    50  
2    60  
Name: Venice, dtype: int64
```

# Pandas

→ *DataFrames*

- This mode still gives you a Series:

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
type(df['Venice'])
```

```
pandas.core.series.Series
```

	Rome	Venice
0	4	40
1	5	50
2	6	60



# Pandas

## → DataFrames

- In order to instead get another DataFrame for a specific column (or more), you can use the `filter()` function, or a bi-dimensional `iloc()` not covered here.

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
print(df.filter(['Venice']))
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

	Venice
0	40
1	50
2	60

# Pandas

## → DataFrames

- In order to instead get another DataFrame for a specific column (or more), you can use the `filter()` function, or a bi-dimensional `iloc()` not covered here.

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
type(df.filter(['Venice']))
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

```
pandas.core.frame.DataFrame
```

# Pandas

## → DataFrames

- To access a row of a DataFrame, you can use the `loc` and/or `iloc` functions, which access “by row”, exactly as for the Series... and returns a Series, in “horizontal”.

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
print(df.loc[0])
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

```
Rome      4  
Venice    40  
Name: 0, dtype: int64
```

# Pandas

## → DataFrames

- To access a row of a DataFrame, you can use the `loc` and/or `iloc` functions, which access “by row”, exactly as for the Series... and returns a Series, in “horizontal”.

```
df = pd.DataFrame({'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
type(df.loc[0])
```

	Rome	Venice
0	4	40
1	5	50
2	6	60

```
pandas.core.series.Series
```

# Pandas

## → DataFrames

- You can also “elect” a data frame column as its index:

```
df = pd.DataFrame({'Quarter': ['q1', 'q2', 'q3'],  
                  'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
df.set_index('Quarter', inplace=True)  
print(df)
```

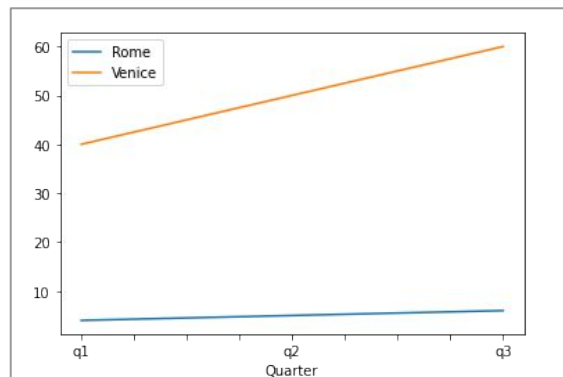
	Rome	Venice
Quarter		
q1	4	40
q2	5	50
q3	6	60

# Pandas

## → DataFrames

- ..and you can plot DataFrames, as the Series and other Pandas data structures.

```
df = pd.DataFrame({'Quarter': ['q1', 'q2', 'q3'],  
                  'Rome': [4, 5, 6],  
                  'Venice': [40, 50, 60]})  
  
df.set_index('Quarter', inplace=True)  
  
plt.plot(df)  
  
plt.show()
```



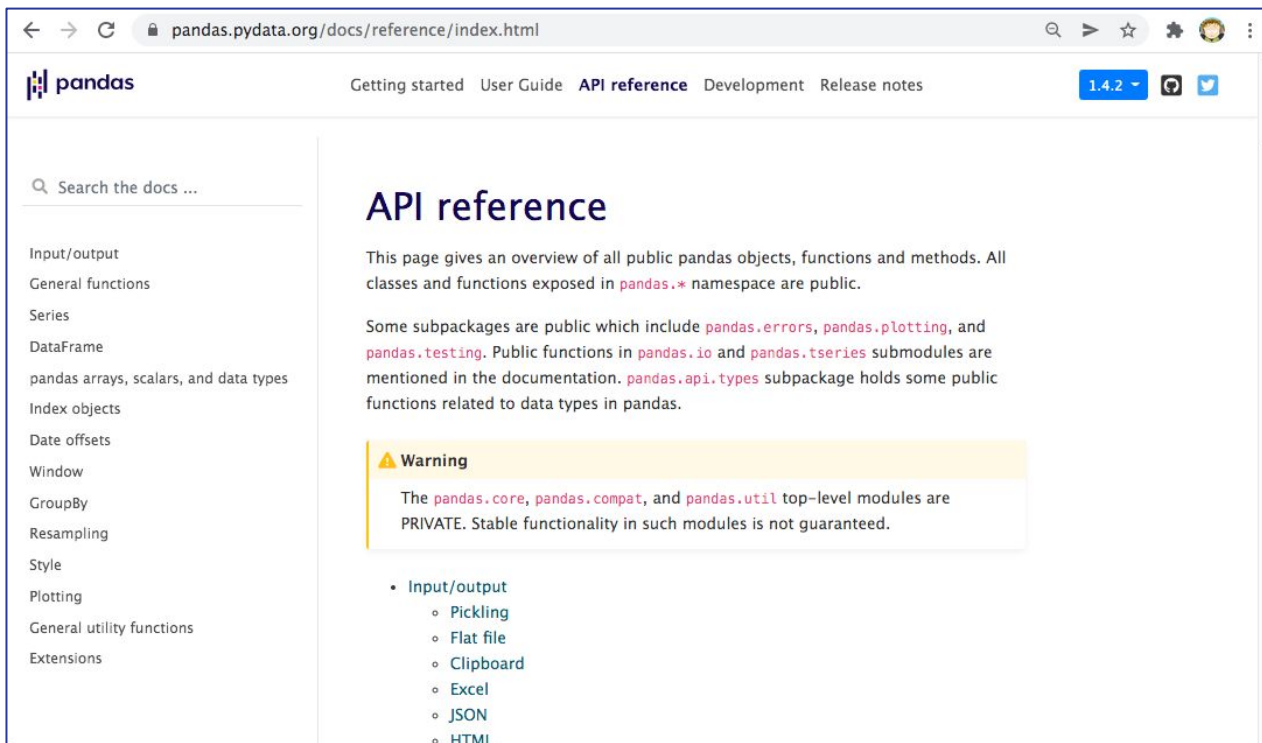
# Pandas

→ *How to read the documentation*

- There are loads of operations which can be done on pandas objects.
- While classic (textbook-like) documentation is always useful, there is another type of documentation that is good to know how to read:
  - the API reference documentation.
- API stands for the Application Programming Interface.
- When you use Pandas, you use its API!

# Pandas

→ *How to read the documentation*



The screenshot shows a web browser window displaying the Pandas documentation website. The URL in the address bar is `pandas.pydata.org/docs/reference/index.html`. The page title is "API reference". The navigation menu includes "Getting started", "User Guide", "API reference", "Development", and "Release notes". The version dropdown is set to "1.4.2".

On the left side, there is a search bar and a list of navigation links: "Input/output", "General functions", "Series", "DataFrame", "pandas arrays, scalars, and data types", "Index objects", "Date offsets", "Window", "GroupBy", "Resampling", "Style", "Plotting", "General utility functions", and "Extensions".

The main content area is titled "API reference" and contains the following text:

This page gives an overview of all public pandas objects, functions and methods. All classes and functions exposed in `pandas.*` namespace are public.

Some subpackages are public which include `pandas.errors`, `pandas.plotting`, and `pandas.testing`. Public functions in `pandas.io` and `pandas.tseries` submodules are mentioned in the documentation. `pandas.api.types` subpackage holds some public functions related to data types in pandas.

**Warning**

The `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are PRIVATE. Stable functionality in such modules is not guaranteed.

- Input/output
  - Pickling
  - Flat file
  - Clipboard
  - Excel
  - JSON
  - HTML



# Pandas

→ *How to read the documentation*

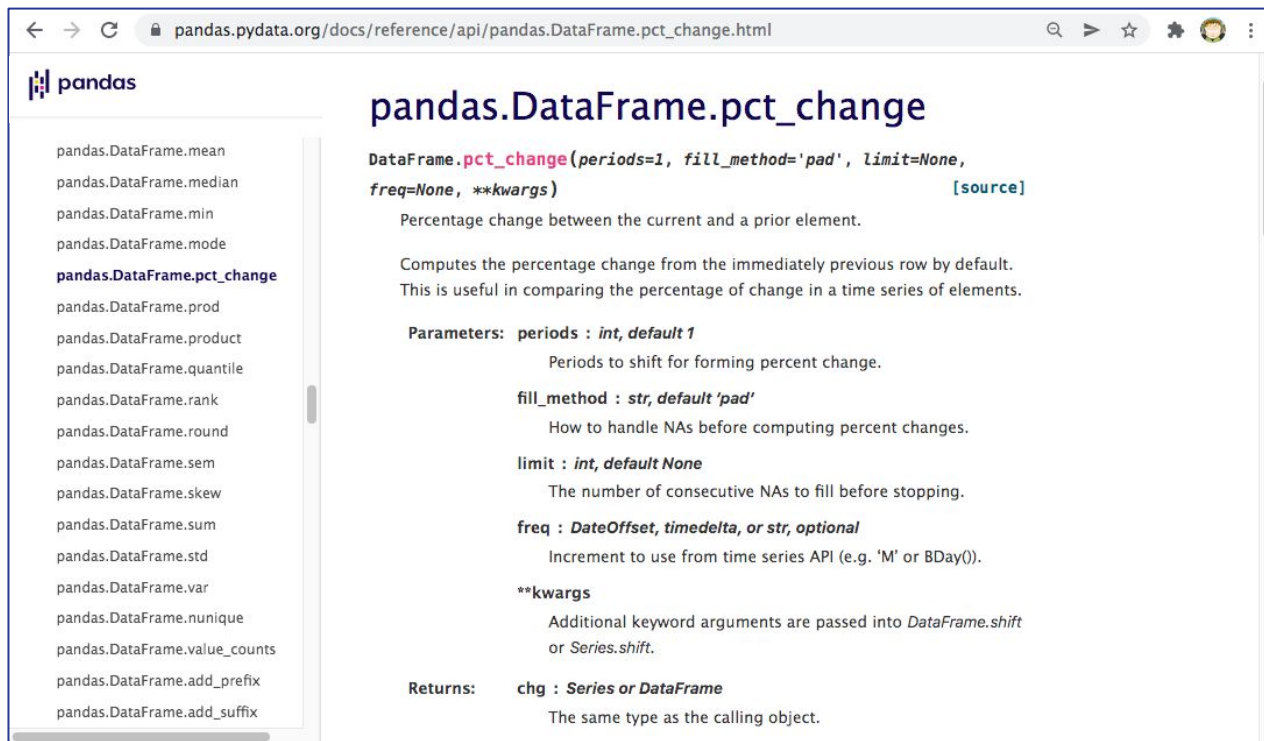
The screenshot shows the Pandas documentation page for DataFrame attributes and underlying data. The page is titled "Attributes and underlying data" and is part of the "API reference" section. The left sidebar contains a list of DataFrame attributes, including `DataFrame`, `DataFrame.index`, `DataFrame.columns`, `DataFrame.dtypes`, `DataFrame.info`, `DataFrame.select_dtypes`, `DataFrame.values`, `DataFrame.axes`, `DataFrame.ndim`, `DataFrame.size`, `DataFrame.shape`, `DataFrame.memory_usage`, `DataFrame.empty`, `DataFrame.set_flags`, `DataFrame.astype`, `DataFrame.convert_dtypes`, `DataFrame.infer_objects`, and `DataFrame.copy`. The main content area lists several attributes and methods:

- Attributes and underlying data**
- Axes**
- DataFrame.index**: The index (row labels) of the DataFrame.
- DataFrame.columns**: The column labels of the DataFrame.
- DataFrame.dtypes**: Return the dtypes in the DataFrame.
- DataFrame.info(verbose, buf, max\_cols, ...)**: Print a concise summary of a DataFrame.
- DataFrame.select\_dtypes(include, exclude)**: Return a subset of the DataFrame's columns based on the column dtypes.
- DataFrame.values**: Return a Numpy representation of the DataFrame.
- DataFrame.axes**: Return a list representing the axes of the DataFrame.

The right sidebar contains a table of contents for the page, including sections like "On this page", "Constructor", "Attributes and underlying data", "Conversion", "Indexing, iteration", "Binary operator functions", "Function application, GroupBy & window", "Computations / descriptive stats", "Reindexing / selection / label manipulation", "Missing data handling", "Reshaping, sorting, transposing", "Combining / comparing / joining / merging", and "Merging".

# Pandas

→ How to read the documentation



The screenshot shows a web browser window displaying the pandas documentation for `DataFrame.pct_change`. The browser's address bar shows the URL `pandas.pydata.org/docs/reference/api/pandas.DataFrame.pct_change.html`. On the left side, there is a sidebar with a list of pandas methods, with `pandas.DataFrame.pct_change` highlighted. The main content area features the title `pandas.DataFrame.pct_change` and the signature `DataFrame.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwargs)` with a `[source]` link. Below the signature, there is a brief description: "Percentage change between the current and a prior element." and a more detailed explanation: "Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements." The "Parameters" section lists: `periods` (int, default 1), `fill_method` (str, default 'pad'), `limit` (int, default None), and `freq` (DateOffset, timedelta, or str, optional). The `**kwargs` section states that additional keyword arguments are passed into `DataFrame.shift` or `Series.shift`. The "Returns" section indicates that the return value is a `Series` or `DataFrame` of the same type as the calling object.

← → ↻ 🔒 pandas.pydata.org/docs/reference/api/pandas.DataFrame.pct\_change.html 🔍 ▶ ☆ ⚙️ 🌐 ⋮

**pandas**

- pandas.DataFrame.mean
- pandas.DataFrame.median
- pandas.DataFrame.min
- pandas.DataFrame.mode
- pandas.DataFrame.pct\_change**
- pandas.DataFrame.prod
- pandas.DataFrame.product
- pandas.DataFrame.quantile
- pandas.DataFrame.rank
- pandas.DataFrame.round
- pandas.DataFrame.sem
- pandas.DataFrame.skew
- pandas.DataFrame.sum
- pandas.DataFrame.std
- pandas.DataFrame.var
- pandas.DataFrame.nunique
- pandas.DataFrame.value\_counts
- pandas.DataFrame.add\_prefix
- pandas.DataFrame.add\_suffix

## pandas.DataFrame.pct\_change

`DataFrame.pct_change`(*periods=1*, *fill\_method='pad'*, *limit=None*, *freq=None*, **\*\*kwargs**) [\[source\]](#)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

**Parameters:**

- periods** : *int*, **default 1**  
Periods to shift for forming percent change.
- fill\_method** : *str*, **default 'pad'**  
How to handle NAs before computing percent changes.
- limit** : *int*, **default None**  
The number of consecutive NAs to fill before stopping.
- freq** : *DateOffset, timedelta, or str*, **optional**  
Increment to use from time series API (e.g. 'M' or BDay()).

**\*\*kwargs**

Additional keyword arguments are passed into `DataFrame.shift` or `Series.shift`.

**Returns:**

- chg** : *Series or DataFrame*  
The same type as the calling object.

# End of part IV

→ *Questions?*

**Next: exercise 4**

# Exercise 4

Let's go through an example together

Try to execute the commands we will see by yourself

First, download the file below and upload it to your Repl.it:

[https://sarusso.github.io/python\\_courses/time\\_series.csv](https://sarusso.github.io/python_courses/time_series.csv)