



# Python - Data Analysis Essentials

David Pinezich

[david.pinezich@gmail.com](mailto:david.pinezich@gmail.com)



## Timeline

Part 1: Introduction, Course objectives, Python basics, Setting up Pycharm, Jupyter, Getting started with numpy theory(array creation, slicing, utility functions) and exercises(puzzles)

Part 2: Continue Numpy theory(concatenating, splitting, universal functions, aggregations, boolean masking, reading and writing data) and exercises(puzzles)

Part 3: Pandas theory(series and dataframe creation, basic dataframe and series methods, data selection, universal functions) and exercises(puzzles)

Part 4: Continue Pandas theory(Reading and writing data, aggregations, filters, groupby) and exercises(finish puzzles, 3 case studies), visualizations using Seaborn, small visualization example of covid



**Universität  
Zürich** <sup>UZH</sup>

IT Training and Continuing Education

# Using Pandas to Get More out of Data



## Learning Objectives

- You know:
  - What a **Series** and **DataFrame** is
  - How to construct a **Series** and **DataFrame** from scratch
  - How to import data using NumPy and/or Pandas
  - How to aggregate, transform, and filter data using Pandas



# Pandas

- Pandas is a **newer** package built **on top of NumPy**
  - Pandas documentation: <https://pandas.pydata.org/pandas-docs/stable/>
- NumPy is very useful for numerical computing tasks
- Pandas allows more flexibility: Attaching labels to data, working with missing data, etc.

```
In [1]: import pandas as pd  
        pd.__version__
```

JUPYTER NB

```
Out [1]: '0.23.4'
```

- *Note:* We are going to use the **pd** alias for the **pandas** module in all the code samples on the following slides



## The Pandas Objects

- Pandas objects are enhanced versions of NumPy arrays: The rows and columns are identified with labels rather than simple integer indices
- **Series** object: A one-dimensional array of indexed data
- **DataFrame** object: A two-dimensional array with both flexible row indices and flexible column names



## The Pandas **Series** Object

- A Pandas **Series** object is a one-dimensional array of indexed data
  - NumPy array: has an *implicitly* defined integer index
  - A **Series** object uses by default integer indices:

```
In [1]: data1 = pd.Series([100,200,300])
```

JUPYTER NB

- A **Series** object can have an *explicitly* defined index associated with the values:

```
In [2]: data2 = pd.Series([100,200,300], index=["a","b","c"])
```

JUPYTER NB

- We can access the index labels by using the **index** attribute:

```
In [2]: d2ind = data2.index
```

JUPYTER NB



## The Pandas **Series** Object

- A Python dictionary maps arbitrary keys to a set of arbitrary values
- A **Series** object maps *typed* keys to a set of *typed* values
  - "Typed" means we know the type of the indices and elements beforehand, making Pandas Series objects much more efficient than Python dictionaries for certain operations
- We can construct a **Series** object directly from a Python dictionary:

```
In [1]: data_dict = pd.Series({"c":123, "a":30, "b":100})
```

JUPYTER NB

- Note: The index for the **Series** is drawn from the sorted keys





## The Pandas DataFrame Object

- A **DataFrame** object is an analog of a two-dimensional array both with flexible row indices and flexible column names
  - Both the rows and columns have a generalized index for accessing the data
  - The row indices can be accessed by using the **index** attribute
  - The column indices can be accessed by using the **columns** attribute



## Constructing DataFrame Objects

- You can think of a **DataFrame** as a sequence of aligned **Series** objects, meaning that each column of a **DataFrame** is a **Series**

```
In [1]: df = pd.DataFrame({"col1":series1, "col2":series2, ...})
```

JUPYTER NB

## Constructing DataFrame Objects

- There are multiple ways to construct a **DataFrame** object
  - From a single Series object:

```
In [1]: pd.DataFrame(population, columns=["population"])
```

JUPYTER NB

- From a list of dictionaries:

```
In [2]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

JUPYTER NB

- From a dictionary of Series objects:

```
In [3]: pd.DataFrame({'population': population, 'area': area})
```

JUPYTER NB

- From a two-dimensional NumPy array:

```
In [4]: pd.DataFrame(np.random.rand(3, 2),  
                    columns=['foo', 'bar'],  
                    index=['a', 'b', 'c'])
```

JUPYTER NB



## Data Selection in Series

- **Series** as a dictionary:
  - Select elements by key, e.g. `data['a']`
  - Modify the **Series** object with familiar syntax, e.g. `data['e'] = 100`
  - Check if a key exists by using the `in` operator
  - Access all the keys by using the `keys()` method
  - Iterate over (column name, Series) pairs by using the `items()` method



## Data Selection in Series

- **Series** as one-dimensional array:
  - Select elements by the implicit integer index, e.g. `data[0]`
  - Select elements by the explicit index, e.g. `data['a']`
  - Select slices (by using an implicit integer index or an explicit index)
    - *Important:* Slicing with an explicit index (e.g., `data['a':'c']`) will *include* the final index in the slice, while slicing with an implicit index (e.g., `data[0:3]`) will *exclude* the final index from the slice
  - Use masking operations, e.g., `data[data < 3]`



## Data Selection in DataFrame

- **DataFrame** as a dictionary of related **Series** objects:
  - Select Series by the column name, e.g. `df['area']`
  - Modify the **DataFrame** object with familiar syntax, e.g. `df['c3'] = df['c2'] / df['c1']`



## Data Selection in DataFrame

- **DataFrame** as two-dimensional array:
  - Access the underlying NumPy data array by using the **values** attribute
    - **df.values[0]** will select the first row
  - Use the **iloc** indexer to index, slice, and modify the data by using the *implicit* integer index
  - Use the **loc** indexer to index, slice, and modify the data by using the *explicit* index



## Ufuncs and Pandas

- Pandas is designed to work with Numpy, thus any NumPy ufunc will work on Pandas **Series** and **DataFrame** objects
- *Index preservation*: Indices are preserved when a new Pandas object will come out after applying ufuncs
- *Index alignment*: Pandas will align indices in the process of performing an operation
  - Missing data is marked with **NaN** ("Not a Number")
  - We can specify on how to fill value for any elements that might be missing by using the optional keyword `fill_value`: **A.add(B, fill\_value=0)**
  - We can also use the **dropna()** method to drop missing values
- *Note*: Any of the ufuncs discussed for NumPy can be used in a similar manner with Pandas objects





## Ufuncs: Operations Between DataFrame and Series

- Operations between a **DataFrame** and a **Series** are similar to operations between a two-dimensional and one-dimensional NumPy array (e.g., compute the difference of a two-dimensional array and one of its rows)



## Checkpoint 1

- Read and run the Pandas notebook until Reading and Writing Data with Pandas
- Solve the Pandas puzzles exercises until exercise 14 (without 14)



**Universität  
Zürich** <sup>UZH</sup>

IT Training and Continuing Education

# Reading (and Writing) Data with Pandas



## File Types

- We will work with *plaintext files* only in this session; these contain only basic text characters and do not include font, size, or colour information
  - *Binary files* are all other file types, such as PDFs, images, executable programs etc.



## The Current Working Directory

- Every program that runs on your computer has a *current working directory*
  - It's the directory from where the program is executed / run
  - *Folder* is the more modern name for a directory
- The *root directory* is the top-most directory and is addressed by `/`
  - A directory `mydir1` in the root directory can be addressed by `/mydir1`
  - A directory `mydir2` within the `mydir1` directory can be address by `/mydir/mydir2`, and so on



## Absolute and Relative Paths

- An *absolute path* begins always with the root folder, e.g. `/my/path/...`
- A *relative path* is always relative to the program's current working directory
  - If a program's current working directory is `/myprogram` and the directory contains a folder `files` with a file `test.txt`, then the relative path to that file is just `files/test.txt`
  - The absolute path to `test.txt` would be `/myprogram/files/test.txt` (note the root folder `/`)



## Reading Data with Pandas

- Pandas provides the `pandas.read_csv()` function to load data from a CSV file (or a file that uses a different delimiter than a comma)
  - The path you specify doesn't have to be on your hard disk; you can also provide the URL to file to read it directly into a Pandas object
  - We can set the optional argument `error_bad_lines` to `False` so that bad lines in the file get omitted and do not cause an error
  - Checkout the documentation to learn more about the optional arguments:  
[https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html)



## Some Interesting Data Sources

- Federal Statistical Office:  
<https://www.bfs.admin.ch/bfs/en/home/statistics/catalogues-databases/data.html>
- OpenData: <https://opendata.swiss/en/>
- OpenData Zurich: <https://www.stadt-zuerich.ch/opendata.secure.html>
- United Nations: <http://data.un.org/>
- World Health Organization: <http://apps.who.int/gho/data/node.home>
- World Bank: <https://data.worldbank.org/>
- Kaggle: <https://www.kaggle.com/datasets>
- Cern: <http://opendata.cern.ch/>
- Nasa: <https://data.nasa.gov/>
- FiveThirtyEight: <https://github.com/fivethirtyeight/data>





## Exporting DataFrame Objects to a File

- We can use the `pandas.DataFrame.to_csv()` method to export a `DataFrame` to a CSV file  
[https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html)
- Overview of all the `DataFrame` methods to import and export data:  
<https://pandas.pydata.org/pandas-docs/stable/api.html#id12>



**Universität  
Zürich** <sup>UZH</sup>

IT Training and Continuing Education

# Aggregating and Grouping Data in Pandas



## Simple Aggregation in Pandas

- As with one-dimensional NumPy array, for a Pandas **Series** the aggregates return a single value
- For a **DataFrame**, the aggregates return by default results within each column
- Pandas Series and **DataFrames** include all of the common NumPy aggregates
  - In addition, there is a convenience method **describe()** that computes several common aggregates for each column and returns the result

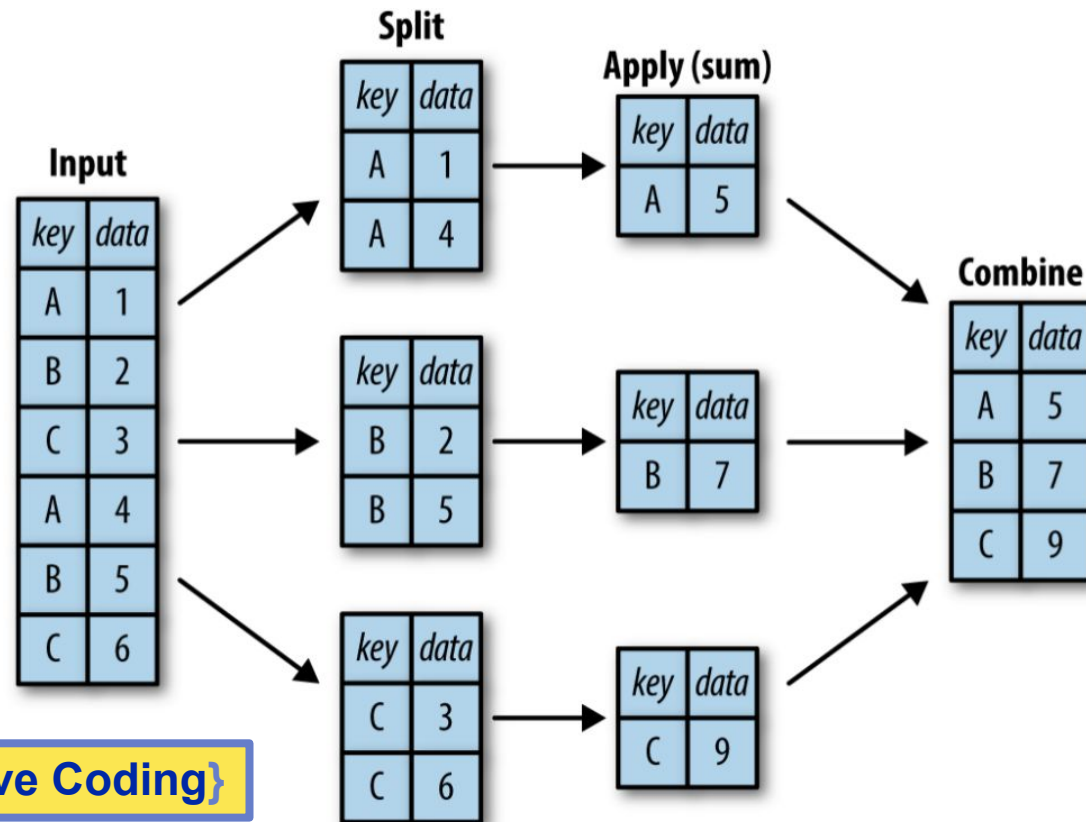


## Split, Apply, Combine

- *Split*: Break up and group a **DataFrame** depending on the value of the specified key
- *Apply*: Apply some function, usually an aggregate, transformation, or filtering, within the individual groups
- *Combine*: Merge the results of these operations into an output array

## Split, Apply, Combine

- Pictured on the right you see an example where in the apply step we use a summation aggregation:
- The `groupBy()` method of `DataFrames` can compute the most basic split-apply-combine operations



Lets check out the `groupBy()` method [{Live Coding}](#)

Source: Python Data Science Handbook



## The GroupBy Object

- The `groupBy()` method returns a `DataFrameGroupBy`: It's a special view of the `DataFrame`
  - Helps get information about the groups, but does no actual computation until the aggregation is applied ("lazy evaluation", i.e. evaluate only when needed)
  - Apply an aggregate to this `DataFrameGroupBy` object: This will perform the appropriate apply/combine steps to produce the desired result
    - You can apply any Pandas or NumPy aggregation function
  - Other important operations made available by a `GroupBy` are *filter*, *transform*, and *apply*



## Column Indexing and Iterating Over Groups

- The **GroupBy** object supports *column indexing* in the same way as the **DataFrame**, and returns a modified **GroupBy** object
- The **GroupBy** object also supports direct iteration over the groups, returning each group as a **Series** or **DataFrame**



## Aggregate, Filter, Transform, and Apply

- *Aggregate*: The `aggregate()` method can compute multiple aggregates at once
- *Filter*: The `filter()` method allows you to drop data based on group properties
  - *Note*: `filter()` takes as an argument a *function* that returns a Boolean value specifying whether the group passes the filtering
- *Transformation*: While aggregation must return a reduced version of the data, `transform()` can return some transformed version of the full data to recombine (meaning that we still have the same number of entries before and after the transformation)
- *Apply*: The `apply()` method lets you apply an arbitrary function to the group results. The function should take a `DataFrame`, and return either a Pandas object or a scalar





## **Checkpoint 2**

- Finish reading and running the Pandas notebook
- Finish the Pandas Puzzle exercises



## Learning Objectives

- You know:
  - What a **Series** and **DataFrame** is
  - How to construct a **Series** and **DataFrame** from scratch
  - How to import data using NumPy and/or Pandas
  - How to aggregate, transform, and filter data using Pandas



**Universität  
Zürich** <sup>UZH</sup>

**IT Training and Continuing Education**

# **Addendum: Working with Files in Python**



## Opening Files with the `open()` Function

- Open a file with the `open()` function by providing a string path indicating the file you want to open
  - The path can be an *absolute* or a *relative* path

```
file = open("/path/to/my/file.txt")
```

CODE

- Typed like this, `open()` will open the file in the *read mode*, meaning we only can read data from the file
- `open()` returns a **File** object, which represents a file on your computer (it's simply another type of value in Python, much like lists and dictionaries)
  - We can now call methods on the **File** object to read its content for example



## Reading the Contents of Files

- We can use the `File` object's `read()` method to read the entire contents of a file as a string value
- Lets assume we have a plaintext file located at `/path/to/file.txt` with `Well, hello there!` as its content. Then:

```
file = open("/path/to/file.txt")  
  
print(file.read())
```

CODE

INTERP.

Content of the file

OUTPUT

## Reading the Contents of Files

- Alternatively, we can use the `File` object's `readlines()` method to get a list of string values from the file, one string for each line of text
- Lets assume we have a plaintext file located at `/path/to/newFile.txt` with the following content:

```
First line  
Second line  
Third line
```

```
file = open("/path/to/newFile.txt")  
print(file.readlines())
```

CODE

INTERP.

```
['First line\n', 'Second line\n',  
 'Third line\n']
```

OUTPUT



## Writing to Files

- We met the *read mode* in the previous slides
- There exist two more modes: the *write mode* and the *append mode*
  - *Write mode* will overwrite the existing file and start from scratch (so watch out!)
    - We pass **"w"** as the second argument to the **open()** function to open the file in write mode
  - *Append mode* will append text to the end of the existing file
    - We pass **"a"** as the second argument to the **open()** function to open the file in append mode



## Writing to Files

- If the filename to `open()` does not exist, both write and append mode will create a new, blank file
- After reading or writing a file, call the `close()` method before opening a file again
- Once we have a file opened in one of the writing modes, we can use the `File` object's `write()` method and pass it a string argument to write it into the file
  - The `write()` method will then return the number of bytes written to the file





## Reader Objects

- We need to create a **Reader** object to read data from a CSV file with the **csv** module
- The **Reader** object lets you iterate over lines in the CSV file



## Reader Objects

```
import csv
```

```
file = open("example.csv")  
exReader = csv.reader(file)  
data = list(exReader)  
print(data)
```

CODE

INTERPRETER

```
[['4/5/2015 13:34', 'Apples', '73'],  
['4/5/2015 3:41', 'Cherries', '85'],  
['4/6/2015 12:46', 'Pears', '14'],  
['4/8/2015 8:59', 'Oranges', '52']]
```

OUTPUT



## Reading Data from Reader Objects in a **for** Loop

- For large files it is disadvantageous to load the entire file into memory at once
- We are going to use the **Reader** object in a **for** loop to iterate over each row of the CSV file, without having to load the entire file into memory
  - *Note:* The **Reader** object can be looped over only once. You must create the **Reader** object anew if you want to reread the CSV file



## Reading Data from Reader Objects in a for Loop

```
import csv

file = open("example.csv")
exReader = csv.reader(file)
for row in exReader:
    print(str(exReader.line_num) + ": " + str(row))
```

CODE

INTERPRETER

```
1: ['4/5/2015 13:34', 'Apples', '73']
2: ['4/5/2015 3:41', 'Cherries', '85']
3: ['4/6/2015 12:46', 'Pears', '14']
4: ['4/8/2015 8:59', 'Oranges', '52']
```

OUTPUT



## Writer Objects

- We can use a **Writer** object to write data to a CSV file
- We can pass a list to the **writerow()** method with the data
  - Each value in the list is placed in its own cell in the output CSV file

```
import csv

file = open("output.csv", "w", newline="")

exWriter = csv.writer(file)
exWriter.writerow(["12/10/2017 14:45", "Fries", "9.5"])
exWriter.writerow(["11/09/2018 10:16", "Bread", "1.2"])

file.close()
```

CODE

output.csv

```
12/10/2017 14:45,Fries,9.5
11/09/2018 10:16,Bread,1.2
```



## The `delimiter` and `lineterminator` Keyword Arguments

- If you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced, we can use the `delimiter` and `lineterminator` keyword arguments with the `reader()` and `writer()` methods
  - The *delimiter* is the character that appears between cells on a row
    - By default the delimiter is a comma `,`
  - The *line terminator* is the character that comes at the end of a row
    - By default the line terminator is a newline

```
import csv

file = open("example.csv")
exReader = csv.reader(file, delimiter="\t", lineterminator="\n")
```



## **Checkpoint 3**

- Finish the Pandas dataset exercises



# Please Save Your Progress





## Questions

- If you have any questions, information, or more about any topic of today's course, feel free to write me at [david.pinezich@gmail.com](mailto:david.pinezich@gmail.com)



## References

- Course content:
  - Al Sweigart, "Automate the Boring Stuff with Python"  
<https://automatetheboringstuff.com/>
  - Jake VanderPlas, "Python Data Science Handbook"  
<https://jakevdp.github.io/PythonDataScienceHandbook/>