Universität
Zürich^UZH

IT Training and Continuing Education

# Python - Data Analysis Essentials

Kaju Bubanja

bubanja.kaju@gmail.com

Universität
Zürich^UZH

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.

JUPYTER NB

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

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:

JUPYTER NB

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

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

{Live Coding}

# 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`

JUPYTER NB

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

# 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

{Live Coding}

# 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)

# 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
  - *Directory* is used in command line contexts, *Folder* is used in graphical user interfaces, they are synonyms
- 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

# 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/
- 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
- Overview of all the `DataFrame` methods to import and export data: https://pandas.pydata.org/pandas-docs/stable/api.html#id12

# 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 `DataFrame`s 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

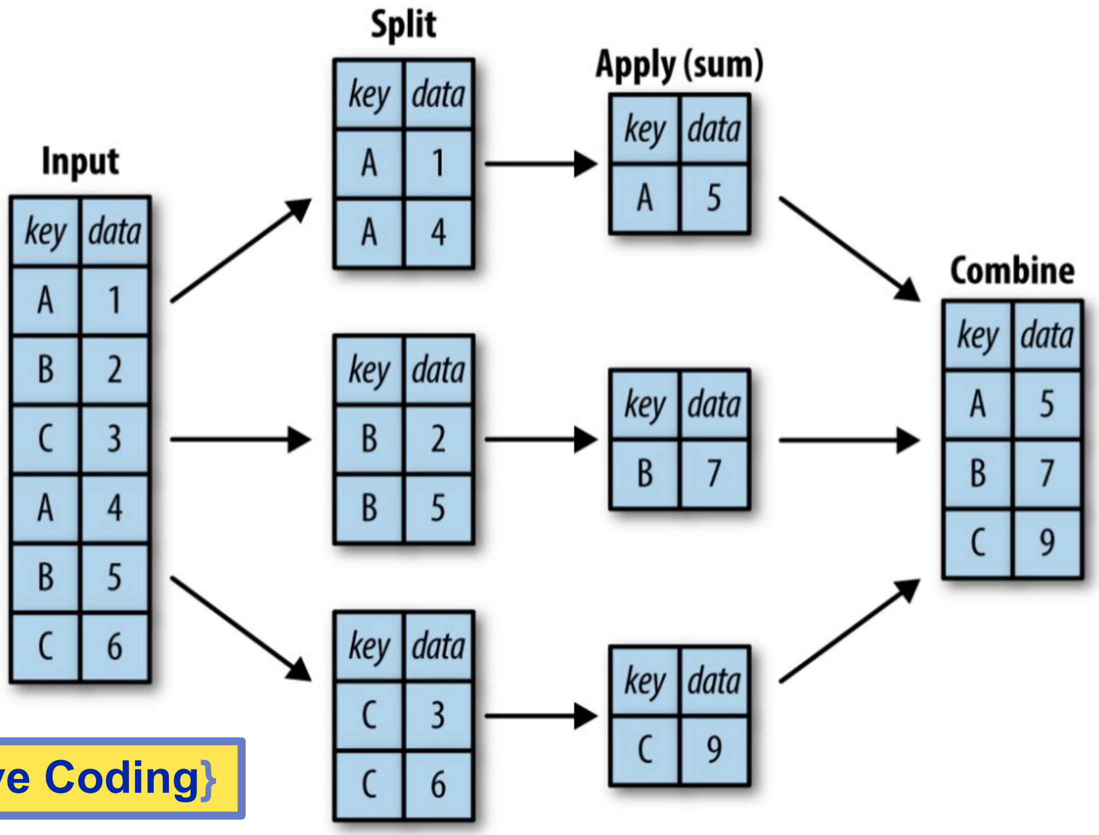
- *Split*: Break up and group a `DataFrame` depending on the value of the specified key
- *Apply*: Apply some function, usually an aggregate, transformation, or filtering, within the individual groups
- *Combine*: Merge the results of these operations into an output array

Source: Python Data Science Handbook

# 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}



Input

| key | data |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |
| A | 4 |
| B | 5 |
| C | 6 |

Split

| key | data |
|---|---|
| A | 1 |
| A | 4 |

| key | data |
|---|---|
| B | 2 |
| B | 5 |

| key | data |
|---|---|
| C | 3 |
| C | 6 |

Apply (sum)

| key | data |
|---|---|
| A | 5 |

| key | data |
|---|---|
| B | 7 |

| key | data |
|---|---|
| C | 9 |

Combine

| key | data |
|---|---|
| A | 5 |
| B | 7 |
| C | 9 |

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

# Checkpoint 3

- Finish the Pandas dataset exercises

# Feedback

- After this course you will receive an email by the course direction asking for feedback about this course
- I would be more than happy to receive as much feedback as possible, since I'd love to further improve the course material and/or my teaching skills where needed
- Constructive criticism and positive comments are both very welcome
  - It's good to know where one can improve, for example by updating the course material or polishing the teaching skills in general
  - It's also good to know which parts of the course and/or which teaching skills helped you the most during the course

# 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/