Python - Data Analysis Essentials

Day 2

Giuseppe Accaputo

g@accaputo.ch

Your Feedback

- Thanks a lot!
- More live-coding: I created notebooks with example codes based on the slides
- Added Pandas exercises to analyse datasets
- In discussion: An intermediate course between the introductory course (APPE*) and this course (APPF*)

Python Data Science Handbook

- Today's course is heavily based on Jake Vanderplas' "Python Data Science Handbook"
- You can find the official online version here: https://jakevdp.github.io/PythonDataScienceHandbook/
- Repository with lots of Jupyter notebooks on the subject: https://github.com/jakevdp/PythonDataScienceHandbook/tree/master/notebooks

Course Outline: Updated

1. A Short Python Primer
2. Data Structures (Lists, Tuples, Dictionaries)
3. Storing and Operating on Data with NumPy
4. Using Pandas to Get More out of Data
5. Addendum: Working with Files in Python

Course Outline: Updated

1. A Short Python Primer
2. Data Structures (Lists, Tuples, Dictionaries)
3. Storing and Operating on Data with NumPy
4. Using Pandas to Get More out of Data
5. Addendum: Working with Files in Python

Universität Zürich[UZH]

IT Training and Continuing Education

Storing and Operating on Data with NumPy

Learning Objectives

- You know:
  - How to create one- and two-dimensional NumPy arrays
  - How to access these arrays
  - How to use the aggregation functions
  - How to work with Boolean arrays

Autosave Your Notebook

– Activate autosave for your current notebook by using `%autosave`:

JUPYTER NB

```
In  [1]: %autosave 30

         Autosaving every 30 seconds
```

NumPy: Numerical Python

- NumPy: Python library that adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays
- NumPy documentation: https://docs.scipy.org/doc/
  - Use your NumPy version number to access the corresponding documentation

JUPYTER NB

```
In  [1]: import numpy as np
         np.__version__

Out [1]: '1.15.4'
```

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

NumPy Arrays

- Python's vanilla lists are heterogeneous: Each item in the list can be of a different data type
  - Comes at a cost: Each item in the list must contain its own type info and other information
  - It is much more efficient to store data in a fixed-type array (all elements are of the same type)
- NumPy arrays are homogeneous: Each item in the list is of the same type
  - They are much more efficient for storing and manipulating data

NumPy Arrays

– Use the `np.array()` method to create a NumPy array:

JUPYTER NB

```
In [1]:  example = np.array([0,1,2,3])
         example

Out [1]: array([1, 2, 3, 4])
```

Multidimensional NumPy Arrays

- *One-dimensional* array: we only need *one coordinate* to address a single item, namely an integer index
- *Multidimensional* array: we now need *multiple indices* to address a single item
  - For an n-dimensional array we need up to n indices to address a single item
  - We're going to mainly work with two-dimensional arrays in this course, i.e. $n = 2$

JUPYTER NB

```
In  [1]: twodim = np.array([[1,2,3],
                            [4,5,6],
                            [7,8,9]])
```

Out [1]:

1	2	3
4	5	6
7	8	9

(Visual aid only, not real output)

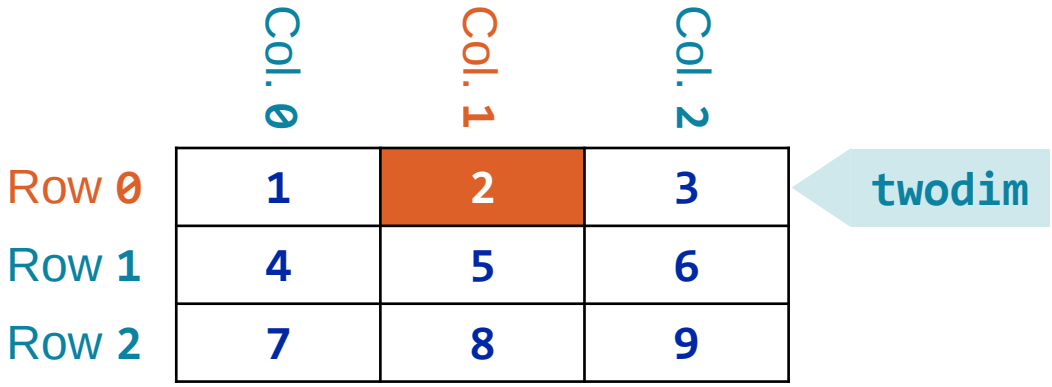
Two-Dimensional NumPy Arrays

– Two-dimensional NumPy arrays have *rows* (horizontally) and *columns* (vertically)

	Column 0	Column 1	Column 2
Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	9

Array Indexing

- Array indexing for one-dimensional arrays works as usual: `onedim[0]`
- Accessing items in a two-dimensional array requires you to specify two indices: `twodim[0,1]`
  - First index is the row number (here `0`), second index is the column number (here `1`)



Lets see how accessing elements works with NumPy arrays, especially with two-dimensional ones **{Live Coding}**

Objects in Python

- Almost everything in Python is an *object*, with its properties and methods
  - For example, a dictionary is an object that provides an `items()` method, which can only be called on a dictionary object (which is the same as a *value of the dictionary type,* or *a dictionary value*)
- An object can also provide *attributes* next to methods, which may describe properties of the specific object
  - For example, for an array object it might be interesting to see how many elements it contains at the moment, so we might want to provide a *size attribute* storing information about this specific property

NumPy Array Attributes

– The type of a NumPy array is **`numpy.ndarray`** (***n*-dimensional array**)

JUPYTER NB

```
In [1]:  example = np.array([0,1,2,3])
         type(example)

Out [1]: np.ndarray
```

– Useful array attributes
  – **`ndim`**: The number of dimensions, e.g. for a two-dimensional array its just 2
  – **`shape`**: Tuple containing the size of each dimension
  – **`size`**: The total size of the array (total number of elements)

Lets create some NumPy arrays and explore the respective attributes **{Live Coding}**

Creating Arrays from Scratch

- NumPy provides a wide range of functions for the creation of arrays: https://docs.scipy.org/doc/numpy-1.15.4/reference/routines.array-creation.html#routines-array-creation
  - For example: `np.arange`, `np.zeros`, `np.ones`, `np.linspace`, etc.
- NumPy also provides functions to create arrays filled with random data: https://docs.scipy.org/doc/numpy-1.15.1/reference/routines.random.html
  - For example: `np.random.random`, `np.random.randint`, etc.

Lets create some NumPy arrays and generate random data **{Live Coding}**

NumPy Data Types

– Use the keyword **dtype** to specify the data type of the array elements:

JUPYTER NB

```
In [1]:  floats = np.array([0,1,2,3], dtype="float32")
         floats

Out [1]: array([0., 1., 2., 3.], dtype=float32)
```

– Overview of available data types: https://docs.scipy.org/doc/numpy-1.15.4/user/basics.types.html

Array Slicing: One-Dimensional Subarrays

- Let `x` be a one-dimensional NumPy array
- The NumPy slicing syntax follows that of the standard Python list:

```
x[start:stop:step]
```

Slice	Description
`x[:5]`	First five elements
`x[5:]`	All elements after index 5
`x[4:7]`	Middle subarray
`x[::2]`	Every other element
`x[1::2]`	Every other element, starting at index 1
`x[::-1]`	All elements, reversed
`x[5::-1]`	Reverses all elements up until index 5 (included)

Array Slicing: Multidimensional Subarrays

– Let `x2` be a two-dimensional NumPy array. Multiple slices are now separated by commas:

```
x2[start:stop:step, start:stop:step]
```

Slice	Description
`x2[:2, :3]`	First two rows and first three columns
`x2[:3, ::2]`	First three rows and every other column
`x2[::-1, ::-1]`	Reverse rows and columns
`x2[:, 0]`	First column
`x2[2, :]`	Third row
`x2[2]`	Same as `x2[2, :]`, so third row again

Lets check out the result of slicing on some concrete examples **{Live Coding}**

Array Views and Copies

- With Python lists, the slices will be *copies*: If we modify the subarray, only the copy gets changed
- With NumPy arrays, the slices will be *direct views*: If we modify the subarray, the original array gets changed, too
  - Very useful: When working with large datasets, we don't need to copy any data (costly operation)
- Creating copies: we can use the `copy()` method of a slice to create a copy of the specific subarray
  - *Note*: The type of a slice is again `numpy.ndarray`

Lets see the effect of *views* and *copies* **{Live Coding}**

Reshaping

– We can use the `reshape()` method on an NumPy array to actually change its shape:

JUPYTER NB

```
In  [1]: grid = np.arange(1, 10).reshape((3, 3))
         print(grid)

         [[1 2 3]
          [4 5 6]
          [7 8 9]]
```

– For this to work, the size of the initial array must match the size of the reshaped array

– *Important*: `reshape()` will return a new view if possible; otherwise, it will be a copy

  – *Remember*: In case of a view, if you change an entry of the reshaped array, it will also change the initial array

Array Concatenation and Splitting

- *Concatenation*, or joining of two or multiple arrays in NumPy can be accomplished through the functions `np.concatenate`, `np.vstack`, and `np.hstack`
  - Join multiple two-dimensional arrays: `np.concatenate([twodim1, twodim2,…], axis=0)`
    - A two-dimensional array has two axes: The first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1)
- The opposite of *concatenation* is splitting, which is provided by the functions `np.split`, `np.hsplit` (split horizontally), and `np.vsplit` (split vertically)
  - For each of these we can pass a list of indices giving the split points

Lets concatenate and split various arrays **{Live Coding}**

Faster Operations Instead of Slow `for` Loops

– Looping over arrays to operate on each element can be a quite slow operation in Python

Lets check this out on a concrete example, which we will be timing using IPython's `%timeit` magic command **{Live Coding}**

– One of the reasons why the for loop approach is so slow is because of the type-checking and function dispatches that must be done at each iteration of the cycle
  – Python needs to examine the object's type and do a dynamic lookup of the correct function to use for that type

NumPy's Universal Functions

- NumPy provides very fast, vectorized operations which are implemented via *universal functions* (ufuncs), whose main purpose is to quickly execute repeated operations on values in NumPy arrays
  - A *vectorized operation* is performed on the array, which will then be applied to each element
- Instead of computing the reciprocal using a for loop, lets do it by using a universal function:

```
In  [1]: %timeit (1.0 / big_array)
```
JUPYTER NB

Lets time this new approach in our Jupyter notebook **{Live Coding}**

- We can use ufuncs to apply an operation between a scalar and an array, but we can also operate between two arrays

```
In  [1]: np.array([4,5,6]) / np.array([1,2,3])
```
JUPYTER NB

NumPy's Universal Functions

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition
`-`	`np.subtract`	Subtraction
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication
`/`	`np.divide`	Division
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `3**2 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Lets see these operators in action **{Live Coding}**

Advanced Ufunc Features: Specifying Output and Aggregates

- ufuncs provide a few specialized features
- We can specify where to store a result (useful for large calculations)
  - If no `out` argument is provided, a newly-allocated array is returned (can be costly memory-wise)

```
In  [1]: np.multiply(x,10, out=y)
```
JUPYTER NB

- *Reduce*: Repeatedly apply a given operation to the elements of an array until only one single result remains
  - For example, `np.add.reduce(x)` applies addition to the elements until the one result remains, namely the sum of all elements
- *Accumulate*: Almost same as reduce, but also stores the intermediate results of the computation

Lets see how these advanced ufunc features work **{Live Coding}**

Some Other Aggregate Functions

Function Name	Description
`np.sum`	Compute sum of elements
`np.prod`	Compute product of elements
`np.mean`	Compute mean of elements
`np.std`	Compute standard deviation
`np.min`	Find minimum value
`np.max`	Find maximum value
`np.argmin`	Find index of minimum value
`np.argmax`	Find index of maximum value
`np.median`	Compute median of elements
`np.percentile`	Compute the qth percentile

Aggregations

- If we want to compute summary statistics for the data in question, aggregates are very useful
  - Common summary statistics: mean, standard deviation, median, minimum, maximum, quantiles, etc.
- NumPy provides fast built-in aggregation function for working with arrays:

JUPYTER NB

```
In  [1]: %timeit np.max(x) # NumPy ufunc
         %timeit max(x)    # Python function
```

- Summing values in an array:

JUPYTER NB

```
In  [1]: %timeit np.sum(x) # NumPy ufunc
         %timeit sum(x)    # Python function
```

Lets check out other aggregation functions {Live Coding}

Multidimensional Aggregates

- By default, each NumPy aggregation function will return the aggregate over the entire array
- Aggregation functions take an additional argument specifying the axis along which the aggregate is computed
  - For example, we can find the minimum value within each column by specifying `axis=0`:

JUPYTER NB

```
In  [1]: twodim.min(axis=0)
Out [1]: array([ … ]) # Array containing min. of each column
```

Lets check out why `axis=0` returns a result in regard to the columns and lets visualize these results by switching between the axes in a two-dim. array

{Live Coding}

Comparison Operators as ufuncs

- NumPy also implements comparison operators as element-wise ufuncs
- The result of these comparison operators is always an array with a Boolean data type:

JUPYTER NB

```
In  [1]: np.array([1,2,3]) < 2
```

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Comparison Operators as ufuncs

– It is also possible to do an element-by-element comparison of two arrays:

```
In  [1]: np.array([1,2,3]) < np.array([0,4,2])
```

JUPYTER NB

These ufuncs will work on arrays of any size and shape.
Lets see an example on how a multidimensional example looks like

{**Live Coding**}

Working with Boolean Arrays: Counting Entries

– The `np.count_nonzero()` function will count the number of `True` entries in a Boolean array:

JUPYTER NB

```
In  [1]: nums = np.array([1,2,3,4,5])
         np.count_nonzero(nums < 4)

Out [1]: 3
```

– We can also use the `np.sum()` function to accomplish the same. In this case, True is interpreted as 1 and False as 0:

JUPYTER NB

```
In  [1]: np.sum(nums < 4)

Out [1]: 3
```

Lets checkout the `np.any()` and `np.all()` functions in relation to Boolean arrays **{Live Coding}**

Working with Boolean Arrays: Boolean Operators

- NumPy also implements bitwise logic operators as element-wise ufuncs
- We can use these bitwise logic operators to construct compound conditions (consisting of multiple conditions)

Operator	Equivalent ufunc
`&`	`np.bitwise_and`
`|`	`np.bitwise_or`
`^`	`np.bitwise_xor`
`~`	`np.bitwise_not`

These ufuncs will work on arrays of any size and shape. **{Live Coding}**
Lets see an example on how a multidimensional example looks like

Boolean Arrays as Masks

- In the previous slides we looked at aggregates computed directly on Boolean arrays
- Once we have a Boolean array from lets say a comparison, we can select the entries that meet the condition by using the Boolean array as a *mask*

x

3	1	5
10	32	100
-1	3	4

x<5

True	True	False
False	False	False
True	True	True

x[x<5]

3	1	5
10	32	100
-1	3	4

▼ (Result)

```
array([3,1,-1,3,4])
```

Lets checkout more examples using this masking operation {Live Coding}

Learning Objectives

- You know:
  - How to create one- and two-dimensional NumPy arrays
  - How to access these arrays
  - How to use the aggregation functions
  - How to work with Boolean arrays

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

Lets inspect the creation and attributes of **`Series`** a bit closer in the notebook **{Live Coding}**

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

Lets see how the resulting `Series` object looks like when we initialize it using a dictionary **{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`**

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

Lets create and examine a specific **`DataFrame`** by using some **`Series`** objects **{Live Coding}**

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

Lets see these creation functions in action **{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
  - Access all the values by using the `items()` method

Lets create a `Series` object and use all the above-mentioned properties to access specific parts of the `Series` **{Live Coding}**

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]`

Lets create another `Series` object and use all the above-mentioned properties to access specific parts of the `Series` **{Live Coding}**

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']`

Lets create a `DataFrame` object and use all the above-mentioned properties to access specific parts of the `DataFrame` **{Live Coding}**

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

Lets create a `DataFrame` object and use all the above-mentioned properties to access specific parts of the `DataFrame` **{Live Coding}**

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

Lets see what index preservation and alignment exactly means on an example **{Live Coding}**

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)

Lets see an example where we first compute the difference between a two-dimensional array and a single row, and then compute the difference between a `DataFrame` and a `Series` {**Live Coding**}

Parsing Data Files with NumPy and 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 NumPy

- We can use the `np.loadtxt()` function to load data from a file
- *Remember:* We can only store elements of a single type in a NumPy array
- Checkout the documentation to learn more about the optional arguments: https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

Lets see some example data and uses of the `numpy.loadtxt()` function **{Live Coding}**

Comma-Separated Values (CSV)

- CSV files are simplified spreadsheets stored as plaintext files
  - Excel for example allows to export spreadsheets as CSV files
- CSV files
  - Don't have types for their values – everything is a string
  - Don't have settings for font size or color
  - Can't specify cell width and heights
  - And more

Comma-Separated Values (CSV)

– Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row:

```
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
```

Source: Automate the Boring Stuff with Python

Reading Data with Pandas

- Pandas provides the `pandas.read_csv()` function to load data from a CSV file
  - The path you specify doesn't have to be on your hard disk; you can also provide the URL to a CSV 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

Lets see how `pandas.read_csv()` works by loading data from different CSV files **{Live Coding}**

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

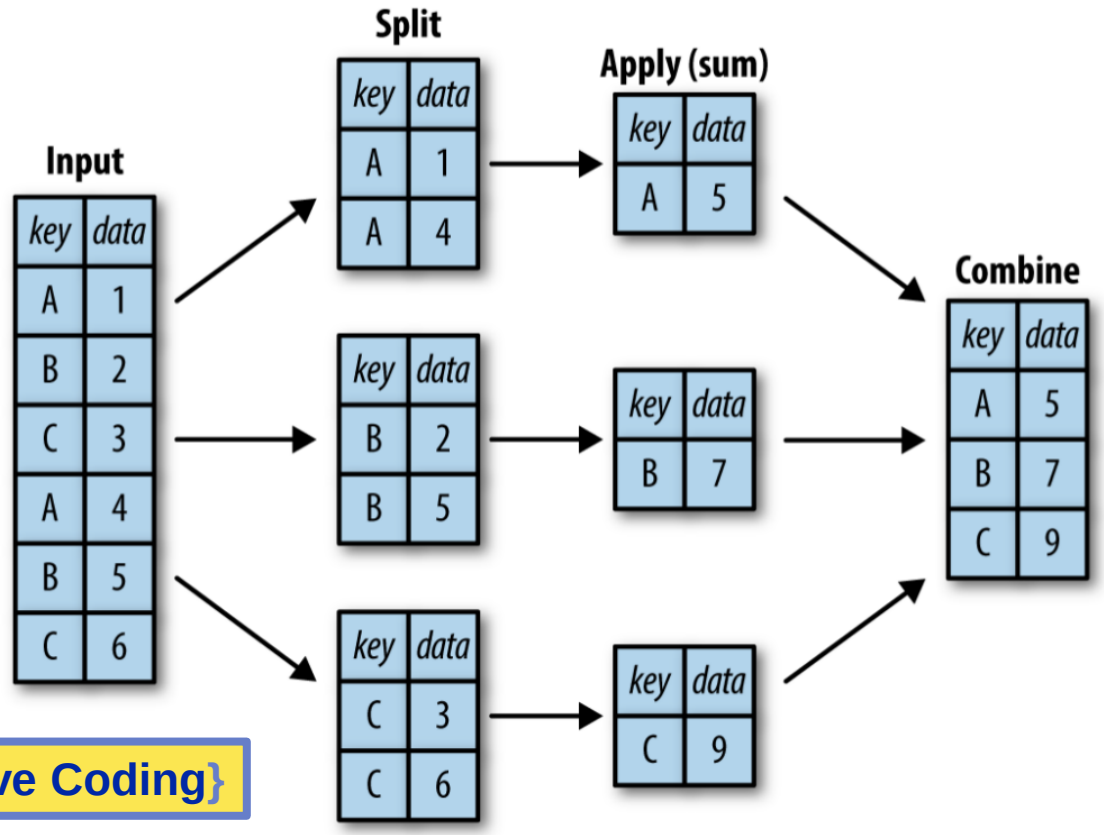
- As with one-dimensional NumPy array, for a Pandas `Series` the aggregates return a single value
- For a `DataFrame`, the aggregates return by default results within each column
- Pandas Series and `DataFrame`s include all of the common NumPy aggregates
  - In addition, there is a convenience method `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

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 `DataFrame`s can compute the most basic split-apply-combine operation



Source: Python Data Science Handbook

Lets check out the `groupBy()` method **{Live Coding}**

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`

Lets check out these `GroupBy` methods **{Live Coding}**

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

Lets check out these additional **`GroupBy`** methods **{Live Coding}**

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[UZH]

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

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

  - 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:

CODE

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

print(file.read())
```

INTERP.

OUTPUT

```
Content of the file
```

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
```

CODE

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

print(file.readlines())
```

INTERP.

OUTPUT

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

Reading the Contents of Files

{Live Coding}

- Lets create a file `test.txt` using a text editor
- Type `Hello, world!` as the content of this text file
- Save it somewhere where you'll find it again, i.e. remember the absolute path to it
- Print the file's content on the screen

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

{Live Coding}

- Lets open a file `write_mode.txt` using the `open()` function in *write mode*
- Write the string `Hello, world!` to it
- Close the file and then open it in *read mode*
- Print the file's content on the screen

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

CODE

```
import csv

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

INTERPRETER

OUTPUT

```
[['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']]
```

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

CODE

```
import csv

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

INTERPRETER

OUTPUT

```
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']
```

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

CODE

```
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()
```

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\n")
```

Please Save Your Progress

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

Questions

– If you have any questions, information, or more about any topic of today's course, feel free to write me at g@accaputo.ch

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/