Universität Zürich UZH

IT Training and Continuing Education

Python - Data Analysis Essentials

Spring Semester 2020, 12.05.2020 – 15.05.2020

Giuseppe Accaputo

g@accaputo.ch

About You

- Your major / occupation
- Your programming experience
- Your goals for this course

About me

- Work
  - Software Engineer, Nexiot AG (since April 2018)
  - Research Assistant, ETH Zürich (2017 – 2018)
  - Teaching Assistant and Course Instructor, ETH Zürich (2014 – 2017)
  - Private Tutor, freelance (2016 – 2018)
  - Software Engineer, LTV Gelbe Seiten AG (2009 – 2011)

- Education
  - B.Sc. & M.Sc. ETH in Computational Science and Engineering (2011 – 2017)
  - B.Sc. FH in Computer Science (2006 – 2009)
  - Vocational education as Systems Engineer (2002 – 2006)

Learning Objectives for This Course

- The main goal is to get a better picture on the essential Python libraries (NumPy and pandas) for preparing, cleaning, transforming and aggregating your data for analysis
- You get Jupyter Notebooks containing the slides' content (one notebook for the NumPy part and one for the pandas part), so you can experiment with all the material at home

Please Feel Free to Always Ask Questions

- Questions are a natural part of the learning process and you're always allowed to ask them
- **Asking questions is an integral part of this course**
- Even if you have a feeling that you're question might "not be good enough," or you don't understand a concept "even if it should be easy to do so," please ask the question nonetheless
  - For one, it gives me the possibility to try and come up with better / clearer explanations
- In case you have any questions after the course, please feel free to contact me at any time via mail at g@accaputo.ch

Learning By Doing (and Making Errors)

- **Programming is best learned by doing**
- Don't be afraid to try stuff out in Python and make errors
  - Errors are a vital part of the learning process and help you understand situations much better

- If you should get stuck on an error during a programming exercise, please always feel free to call for my help or the help of fellow students
- Also, don't be afraid to use pen and paper to solve the exercises or when you are trying to understand a specific concept
  - For one, it helps a lot to step away from the computer from time to time
  - It also helps a lot to write down the immediate steps when trying to understand a complicated concept

Feedback

- This is the second installment of this course
- I'm very thankful for all the feedback I get (be it positive or negative), since I want you to feel comfortable and I love to improve my courses and my teaching skills
  - Course is moving too fast?
  - I'm not speaking clearly enough?
  - Please feel free to inform me about anything whenever you feel like it ☺

Course Outline

1. A Very Short Introduction to Jupyter Notebooks
2. Important Basics of the Python Programming Language
3. Storing and Operating on Data with NumPy
4. Using Pandas to Get More out of Data
5. Addendum: Working with Files in Python

A Very Short Introduction to Jupyter Notebooks

Python Code Is Portable

- Python code can be interpreted and run / executed using any current operating system, e.g. Windows, OS X, and Linux

The Python Ecosystem Is Huge

- Python already comes with a lot of useful tools and libraries
- Nonetheless, there also exist thousands of third-party modules and libraries which can be used to accomplish various tasks, NumPy and Pandas being just two of them
  - https://awesome-python.com/

Jupyter Notebook

- Interactive computing with Python
- Offers introspection: We can inspect values and errors, time our functions, and more
- Offers tab completion and history
- Offers an interface with support for code, text, mathematical expressions and more

Help and Documentation

- How do I call a function? What arguments and options does It have?
- What does the source code of this Python value / object look like?
- What is in this package I imported?
- What variables / attributes or methods does this value / object have?

Help and Documentation

- We can access documentation with **?**

IPYTHON

```
In [1]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
```

- This notation works for about anything, including object methods and functions (as we will see later)

Shell Commands

- The shell is a way to interact textually with your computer
  - Operating systems existed long before graphical user interfaces as we know and use today
- We can create folders, files, copy and delete them, and more with a shell
  - Basically, we can submit a lot of commands via shell to the computer

Shell Commands

- `pwd`: Print the working directory (where we currently are in the file system)
- `ls`: List working directory contents
- `cd`: Change directory
- `mkdir`: Make new directory

Running External Code with `%run`

– We can use a text editor to write code and run it with `%run`

print.py

```
def fun(lst):
  for e in lst:
    print(e)

fun([1,2,3,4])
```

IPYTHON

```
In [1]: %run print.py
1
2
3
4
```

Important Basics of the Python Programming Language

(…at least for this course)

Values and Data Types

- *Values* are fundamental things like the number `2` or `1.234`, or the string `Hello`
- A *data type* is a category for values, and a value always belongs to a single data type
  - Integer data type: `-1, -100, 0, 12, 34`
  - Float data type: `-1.324, 0.14123, 10.1, 100.0`
  - String data type: `‘Hello’, ‘Word’, ‘Spaces are included’`
  - List data type: `[1,2,3,4]`
  - Tuple data type: `("A", "B", "C")`
  - Dictionary data type: `{"k1": 1, "k2": 132}`

The List Data Type

1. Initialization of a list: (*Note*: A list can contain elements of different data types)

CODE

```
lst = ["one", "two", 3, 4, 5]
```

2. Accessing elements: (*Note*: First element in the list is at the index `0`)

CODE

```
el1 = lst[0]
eln = lst[-1]
```

3. Changing values: (*Note*: A Python list is a *mutable* data structure)

CODE

```
lst[0] = "abc"
lst[4] = 423.132
```

The List Data Type

4. Accessing slices: (*Note*: The slice goes up to, but will not include, the value at the second index)

CODE

```
sl1 = lst[2:3]
sl2 = lst[1:]
```

5. Removing elements: (*Note*: Removing an element changes the underlying list structure)

CODE

```
del lst[2]
```

6. Iterating over a list's elements:

CODE

```
for el in lst:
  print(el)
```

7. Check if a value exists in a list:

CODE

```
val_exists = "one" in lst
```

The Tuple Data Type

1. Initialization of a tuple: (*Note*: A tuple can contain elements of different data types)

CODE

```
tpl = (1, 2, 3, "four", 5)
```

2. Accessing elements: (*Note*: First element in the tuple is at the index `0`)

CODE

```
t1 = tpl[0]
eln = tpl[-1]
```

3. We cannot change elements of a tuple, since it's an *immutable* data structure.
   What we can do instead is copy its elements into a mutable data structure:

CODE

```
lst = list(tpl)
lst[0] = 34
lst[4] = "abc"
```

The Tuple Data Type

4. Accessing slices: (*Note*: The slice goes up to, but will not include, the value at the second index)

CODE

```
sl1 = tpl[2:3]
sl2 = tpl[1:]
```

5. We cannot remove elements from a tuple, since it's an *immutable* data structure.

6. Iterating over a tuple's elements:

CODE

```
for el in tpl:
  print(el)
```

7. Check if a value exists in a tuple:

CODE

```
val_exists = 1 in tpl
```

The Dictionary Data Type

1. Initialization of a dictionary: (*Note*: all keys must be of the same data type; values can be *anything*)

CODE

```
dct = {"k1": "v1", "k2": "v2"}
```

2. Accessing values: (*Note*: We access a value by its corresponding key)

CODE

```
v1 = dct["k1"]
v2 = dct["k2"]
```

3. Changing values: (*Note*: A Python dictionary is a *mutable* data structure)

CODE

```
dct["k1"] = "v1new"
```

The Dictionary Data Type

4. Accessing slices is not possible, since the data type of the key is not always integer

5. Removing elements: (*Note*: Removing an element changes the underlying list structure)

CODE

```
del dct["k1"]
```

4. Iterating over a list's key-value pairs:

CODE

```
for (k,v) in dct.items():
  print(k, ": ", v, sep="")
```

5. Check if an entry exists for a specific *key*:

CODE

```
entry_exists = "k1" in dct
```

Dictionaries vs. Lists

- Lists are ordered
  - First item in a list is located at the index 0
  - We can slice lists
  - Trying to access an index that is out of range results in an error message
- Dictionaries are unordered
  - There is no "first" item, since we can only access items using keys
  - We cannot slice dictionaries
  - Trying to access a key that does not exist results in an error message

Dictionaries vs. Lists

– Lists are ordered; the order of the elements matters:

CODE

```
l1 = [1,2,3,4]
l2 = [2,1,3,4]

print(l1 == l2)
```

INTERP.

OUTPUT

```
False
```

– Dictionaries are unordered; the order of the elements does not matter:

CODE

```
d1 = {"a":13, "b":14}
d2 = {"b":14, "a":13}

print(d1 == d2)
```

INTERP.

OUTPUT

```
True
```

Methods

Learning Objectives

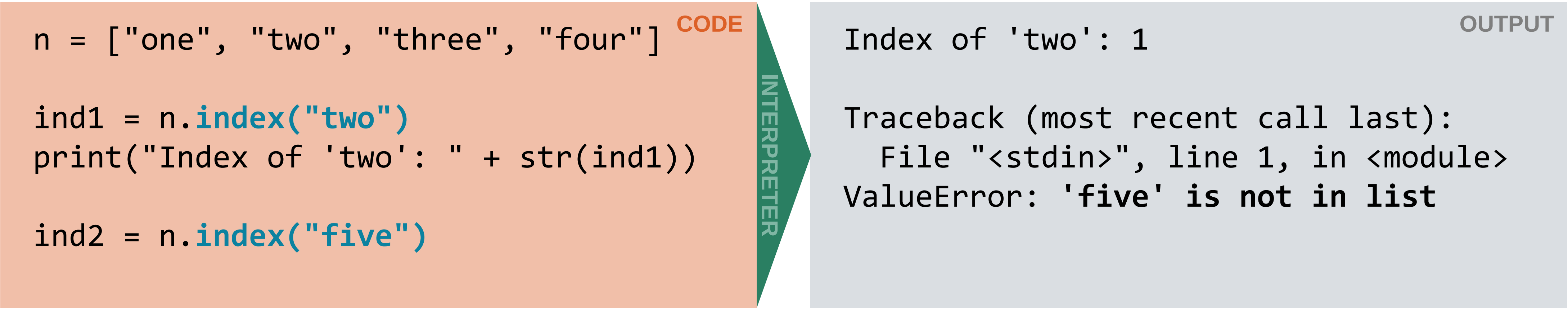
- You know
  - how to call a method
  - how to use tab-completion to help you with methods
  - that different data types may provide different methods

Methods

- A *method* is the same thing as a function, except it is called on a value
  - Function call: `my_fun(a,b,c)`
  - Method call: `my_list.index("k")`
    - We called the `index` method on the value of `my_list`, which is of type `list`
- Each data type (`str`, `list`, `dict`, etc.) has its own set of methods
  - The `list` data type has several useful methods for finding, adding, removing, and manipulating values in a list
- A method always acts on the value it has been called on
  - `list1.index("k")` → `index("k")` acts on the value of `list1`
  - `list2.index("e")` → `index("e")` acts on the value of `list2`

Finding a value in a List: The `index()` Method

– The list data type provides an `index()` method, to which we can pass a value. If that value exists in the list, the index of the value is returned, else Python produces a `ValueError` error



```
n = ["one", "two", "three", "four"]

ind1 = n.index("two")
print("Index of 'two': " + str(ind1))

ind2 = n.index("five")
```

```
Index of 'two': 1

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'five' is not in list
```

In-Place Changes

- Both the `append()` and `insert()` methods will change the list on which they're called on
- We call these kind of changes *in-place changes*

Adding Values to a List: The `append()` and `insert()` Methods

- Lets add a new element at index **1** of the list:

CODE

```
alpha = ["a", "b", "c"]

alpha.insert(1, "w")

print(alpha)
```

- *Note*: After adding the new element, all previously existing elements at index 1, 2, and above are moved to the right. This can be a costly operation if we insert elements in very large lists like this

Different Methods for Different Data Types

- Methods belong to a single data type
  - `append()` and `insert()` are list methods and can be called only on lists, not on other values such as strings or integers

CODE

```
num = 1023

# What might happen here?
num.insert(1, "w")
```

Removing Values from Lists (In-Place): The `remove()` Method

- We can pass a value we want to be removed to the `remove()` method of a specific list:

CODE

```
alpha = ["a", "b", "c"]

alpha.remove("a")

print(alpha)
```

- *Note:* If you know the index of the value we want to remove, we can still use the `del` operator for the removal; if you know the value, just use the `remove()` method

Sorting the Values in a List (In-Place): The `sort()` Method

– We can sort lists of strings or numbers by calling the `sort()` method on a specific list:

CODE

```
alpha = ["c", "a", "b"]
alpha.sort()
print(alpha)

num = [3.14, 10, 1, -23, 0.4]
num.sort()
print(num)
```

INTERPRETER

OUTPUT

```
['a', 'b', 'c']
[-23, 0.4, 1, 3.14, 10]
```

Learning Objectives

- You know
  - how to call a method
  - how to use tab-completion to help you with methods
  - that different data types may provide different methods

Storing and Operating on Data with NumPy

Python Data Science Handbook

- The 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

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
  - How to read and write files with NumPy

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)

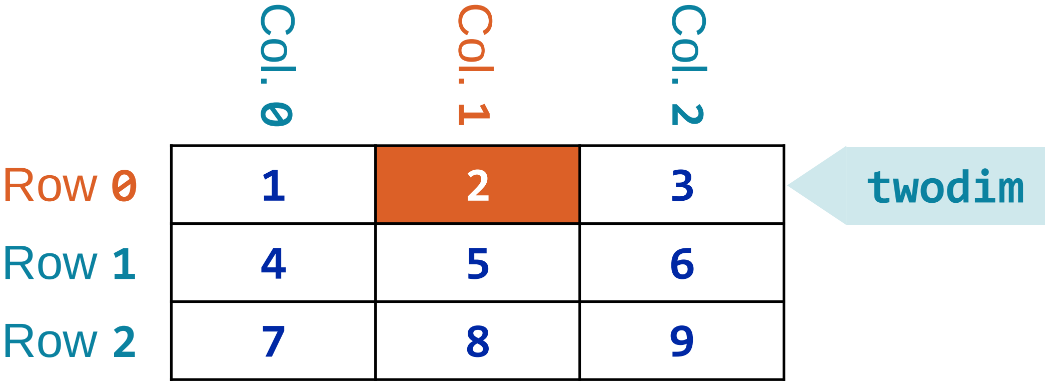
Two-Dimensional NumPy Arrays

- 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`)



	Col. 0	Col. 1	Col. 2
Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	9

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)

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.

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

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

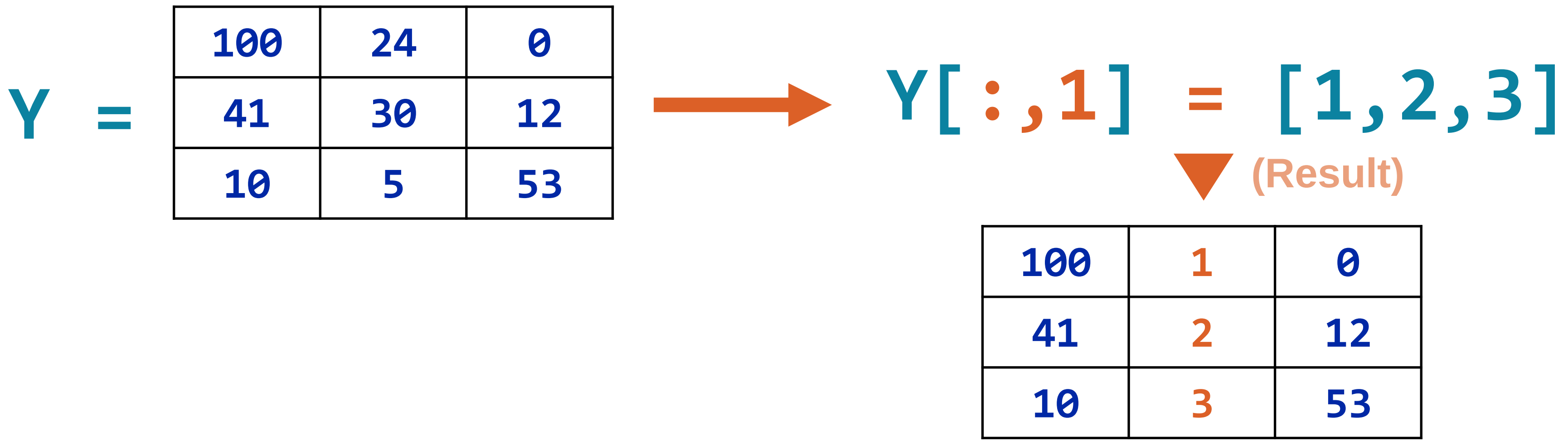
Slice	Description
`Y[:2, :3]`	First two rows and first three columns
`Y[:3, ::2]`	First three rows and every other column
`Y[::-1, ::-1]`	Reverse rows and columns
`Y[:, 0]`	First column
`Y[2, :]`	Third row
`Y[2]`	Same as `Y[2, :]`, so third row again

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)
  - **Important:** If we use an index array (which we will see later) to select data it will return a copy of the data
- 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`

Array Slicing: Multidimensional Subarrays

- Since we're working with direct views, we can update the data using array slicing:



Y =

100	24	0
41	30	12
10	5	53

```
Y[:,1] = [1,2,3]
```

(Result)

100	1	0
41	2	12
10	3	53

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

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 the `%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`)

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)

JUPYTER NB

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

- *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}**

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:

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

JUPYTER NB

- Summing values in an array:

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

JUPYTER NB

Lets check out other aggregation functions {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

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}

The Boolean Data Type

- Boolean data type: `True`, `False` (only two possible values)
- *Comparison operators* compare two values and evaluate to a single Boolean value
  - The comparison operators are `==`, `!=`, `<`, `>`, `<=,` and `>=`
- *Boolean operators* are used to compare Boolean values
  - The Boolean operators are `or`, `and`, and `not`
- We can mix Boolean and comparison operators to create *conditions*

{Live Coding}

- Lets see the Boolean and comparison operators in action

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:

JUPYTER NB

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

{Live Coding}

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

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

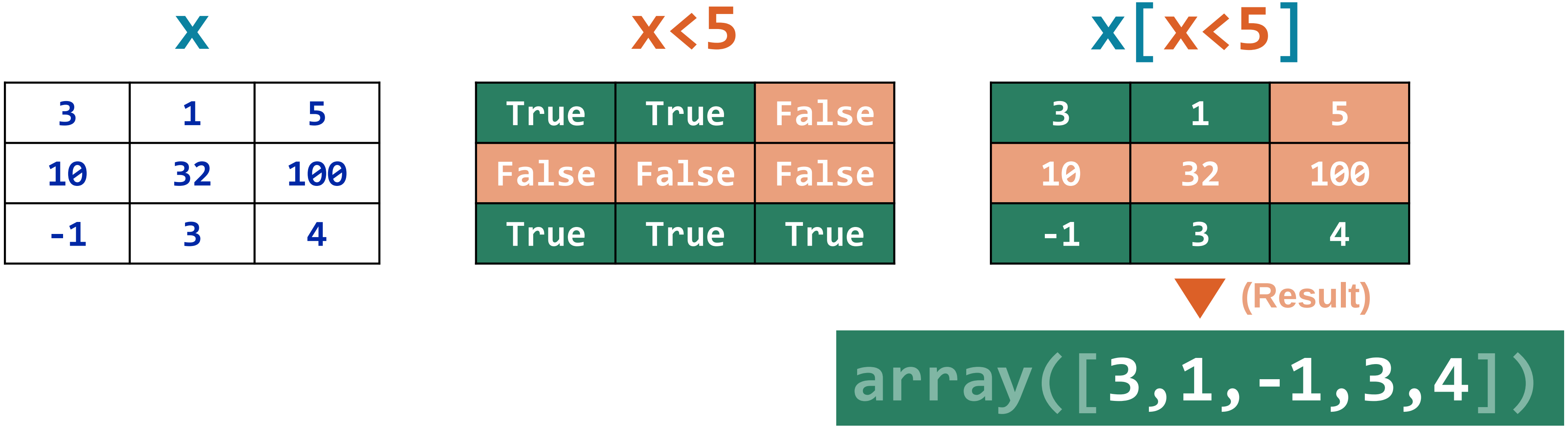
- NumPy also implements bitwise logic operators as element-wise ufuncs
- We can use these bitwise logic operators to construct compound conditions (consisting of multiple conditions)

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.
Lets see an an example on how a multidimensional example looks like {Live Coding}

Boolean Array Indexing

- We can use a Boolean array to select specific entries of an array



`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])
```

- **Please note:** Only slices return a view; all other index array accesses return copies. This means that `values = x[x<5]` will be a *copy* of the data, whereas `values = x[:,:1]` will be a *view*.

Reading and Writing Data with NumPy

- We can use the `np.savetxt()` function to save NumPy data to a file
- 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 *normal* NumPy array
- Use the shell commands `!ls`, `!pwd`, and `!cd` to navigate the file system if necessary

Lets checkout how we can read and write files with NumPy **{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 CSV Data with NumPy

- Some CSV data contains a mix between numbers and strings, or might have missing values
- We can use the `np.genfromtxt()` function to load mixed data from such a file into a NumPy array

Lets import the FIFA 2019 CSV file using `numpy.genfromtxt()` **{Live Coding}**

Dataset source: https://www.kaggle.com/karangadiya/fifa19

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
  - How to read and write files with NumPy

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:

JUPYTER NB

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

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

JUPYTER NB

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

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

JUPYTER NB

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

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),                              JUPYTER NB
                      columns=['foo', 'bar'],
                      index=['a', 'b', 'c'])
```

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

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)

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

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

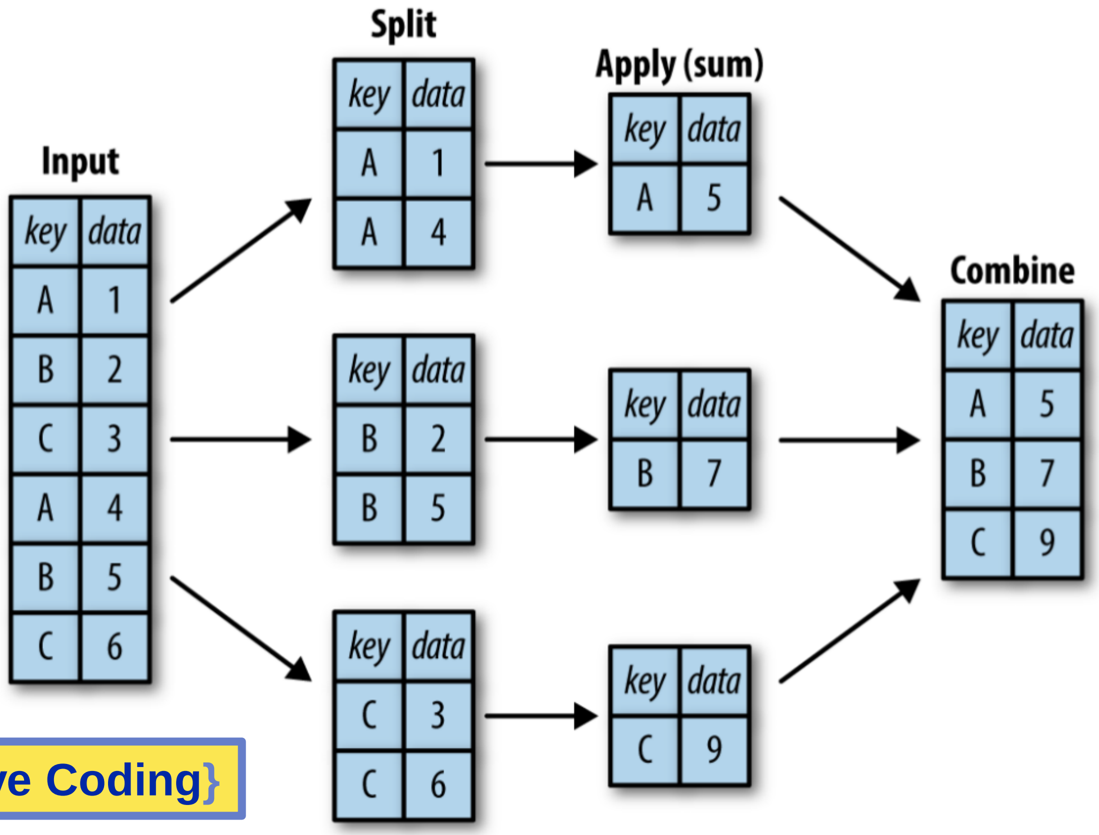
Split, Apply, Combine

- *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 `DataFrame`s returns an object with which we can further run the apply and/or combine steps



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`
  - Stores 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

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

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/

Questions

- If you have any questions, information, or more about any topic presented in this course, feel free to contact me at g@accaputo.ch