Chapter 1

Python pandas

Prof. Dr. K. Verbarg

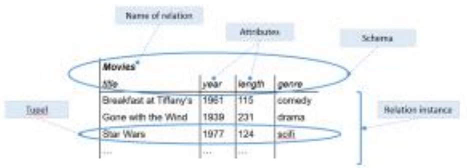
Outline

Introduction

About our motives and Python basics

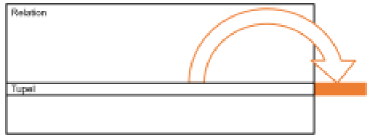
Schema

The Python pandas basic data structures DataFrame, Series and Index



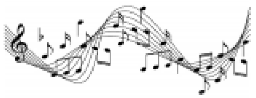
Projection

Working on columns and computing new columns



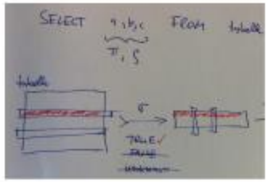
Interlude

Some remarks not fitting elsewhere



Selection

Selection of rows



pd.NA

Working with NULL values

SQL	evaluates to
`1 + 2`	3
`1 + NULL`	NULL
`3 * NULL`	NULL
`1 > NULL`	UNKNOWN
`1 = 1`	TRUE
`42 = NULL`	UNKNOWN
`NULL = NULL`	UNKNOWN
`NULL IS NULL`	TRUE
`(1=1) AND (1=1)`	TRUE
`(1=1) AND (1=NULL)`	UNKNOWN

NOT(A)

A	¬A
F	T
U	U
T	F

AND(A, B)

A ∧ B		B: F	B: U	B: T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

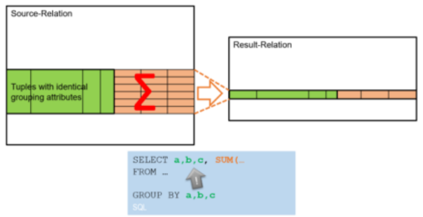
OR(A, B)

A ∨ B		B: F	B: U	B: T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

© wikipedia, "Kleene three-valued logic".

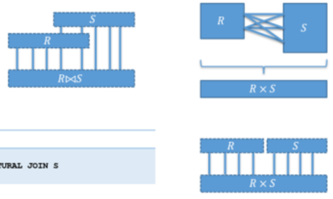
Grouping and aggregation

We look at all tuples having the identical values for all grouping attributes and make one new tuple for them.
Each non-grouping attributes can be aggregated to one single value (per group).



Joins

Combining DataFrames



Relational algebra	SQL
R ⋈ S	SELECT … FROM R **NATURAL JOIN S** WHERE …
R ⋈<condition> S	SELECT … FROM R **JOIN S ON** <condition> WHERE …
R ⟕<condition> S	SELECT … FROM R **LEFT OUTER JOIN S ON** <condition> WHERE …

RDBMS versus pandas

Introduction

About our motives and Python basics

The requirements list for RDBMS

1) Define the structure of the data (schema) by a data-definition language (DDL)
2) Manipulate and query the data using a data-manipulation language (DML)
   → this should be convenient (by using a special language)
   Example in SQL (pronounced “sequel”: this is the standard for RDBMS):
   `SELECT balance FROM Accounts WHERE accountNo = 123456;`
3) Support large amounts of data over a long period of time
   → access to the data should be efficient (fast)
4) Enable durability or persistence: data should be safe (a “bank”) and recoverable in case of failures, errors or misuse.
5) Many users access data at once being protected against unexpected interactions (called isolation).
6) Reliability: The system is available 99.999…% of the time

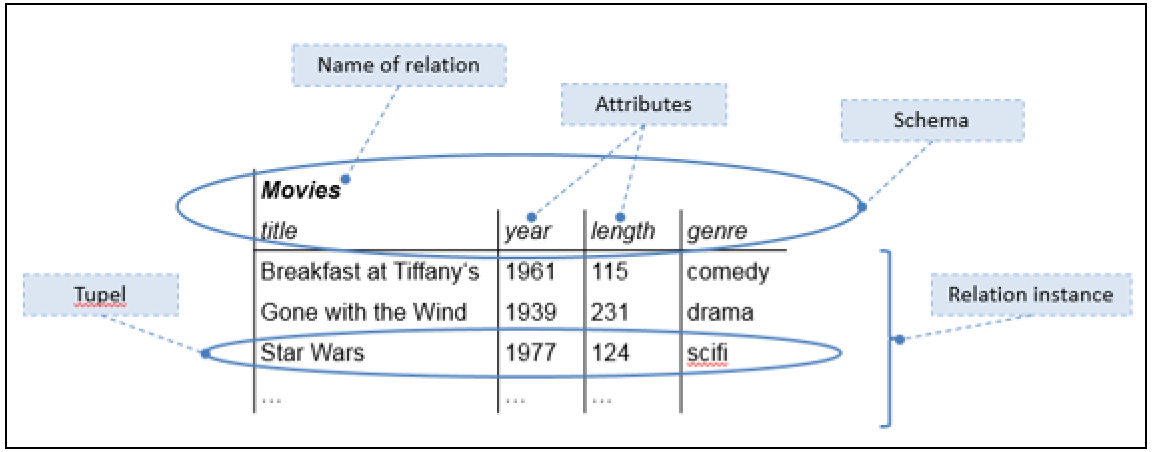
What do we need for analysing static datasets (just once maybe)?

1) Would be nice, but pandas will try to infer the schema automatically from data.
2) Yes, we will see the pandas-counterparts for SQL commands. However, pandas provides a more functional approach.
3) Yes, but this may be an issue when we have all data in the RAM.
4) No
5) No (especially not transactions)
6) No

RDBMS – relational algebra

A <u>relation</u> is a relation schema (name of the relation plus a list of attributes) plus the data stored in the relation (a bag or multiset of tuples).

Example:



Basic commands of relational algebra:

- Projection (π – “pi“): Select only some of the columns
- Selection (σ – “sigma“): Select only some of the rows
- Renaming (ρ – “rho“): The instance is the same, only the schema changes
- Duplicate elimination (δ – “delta“)
- Grouping and aggregation result (γ – “gamma“)

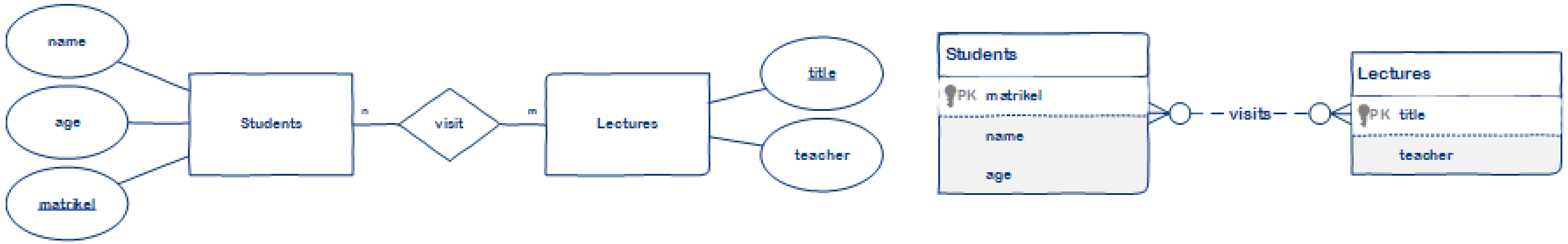
RDBMS – SQL

- CREATE TABLE with primary, foreign key and check constraints
- SELECT with GROUP BY / ORDER BY / Subqueries
- Joins (Cartesian product, natural join, left outer join, theta join, equijoin)

> We can compute $R \times S$ with the nested loop algorithm:
>
> ```
> for i = 1 to n := |R|
>    for j = 1 to m := |S|
>         print R[i],S[j]
> ```
>
> The runtime is $O(n^2)$, assuming that $n > m$ (otherwise exchange R and S).

- INSERT / DELETE / UPDATE
- NULL arithmetic
- Standard functions in the SELECT column list (e.g. string concatenation with “||”)
- Views

Entities, relations, attributes, keys, multiplicity with Chen or UML-crowfoot notation.



Mapping of ERM to relations.

Learning database basics

If you want to refresh your DB knowledge, this is a good reference.

It is available in our library as online PDF (accessible from within the HOST network)

https://ebookcentral.proquest.com/lib/hs-stralsund/detail.action?docID=6363308

Motives for having alternatives to RDBMS

RDBMS proved to be highly beneficial for many applications. A RDBMS is "automatically" efficient and scales well to large dataset (on disk). SQL provides a simple declarative, yet extremely powerful toolset to query and maintain data.

But it comes with a large overhead to provide the guarantees like ACID (atomicity, consistency, isolation, durability). Operating a RDBMS requires considerable effort and cost (licensing, administration, hardware).

Therefore, for specific applications various alternative approaches were developed.

- Think of replacing relations with other data structures like JSON documents, key/value pairs, graphs, linked data (NoSQL databases)
- distributed databases for better scalability in exchange for absolute consistency
- unstructured data
- stream data from sensors

To quickly develop one-time statistical evaluations, data science applications, providing a data basis for machine learning…

… we look into **Python pandas** now.

Ironically, when such pandas projects grow, performance and memory limitations may get an issue again.

What we want to do now

There is some discussion indeed, whether Python pandas or **R** plus **tidyverse** / **dyplr** is the best solution.

If you are interested, look into
https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/

For an overview of dyplr operations, see
https://github.com/rstudio/cheatsheets/blob/master/data-transformation.pdf

We leave this discussion to the reader. It may depend on your background and the ecosystem you are familiar with (R for statistics, Python for programmers or machine learning). You might even want to mix the approaches.

Reasons to go with Python pandas here are:

- Python is a very easy (to learn) programming / script language. It provides very powerful functions and interfaces so that we are completely equipped.
- If you want to extend data flows in SAP Data Warehouse Cloud, you need Python pandas (and not SAP ABAP).

Our goals are:

- review the very basics of Python to be able to proceed
- working with relational data in pandas along the lines of what we know from SQL commands
- input and output of data
- data cleansing and basic descriptive statistics

Disclaimer: In contrast to SQL tool set, in pandas there are many competing solutions to a problem and lots of very specific functions. We aim to show one way through.

In your VM, the **anaconda** distribution (cost-free individual edition) is installed and necessary Python packages activated in the standard Environment "base".

You can install new packages with the "Anaconda Prompt".

The command used is `conda`.

To update conda itself, first do:

```
conda update conda
```

To update all installed packages, do:

```
conda update --all
```

Disclaimer: There is a high rate of progress in this area.

Anaconda Prompt (Anaconda3)

App

Anaconda Prompt (Anaconda3)

```
(base) C:\Users\verbarg>conda update conda
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\verbarg\Anaconda3

  added / updated specs:
    - conda


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    qtconsole-5.1.1            |     pyhd3eb1b0_0          98
    tk-8.6.11                  |       h2bbff1b_0         3.3
    traitlets-5.1.0            |     pyhd3eb1b0_0          89
    ------------------------------------------------------------
                                           Total:         3.4
```

JupyterLab

We will not use the **JupyterLab** notebook editor.

We prefer **VSCode** with Extensions Python / Jupyter.

https://code.visualstudio.com/docs/datascience/jupyter-notebooks

A notebook (`.ipynb` file) is attached to a kernel (a Python process). Unless you restart the kernel, it keeps its current state (values of variables).

The notebook consists of cells containing code. Cells can be executed one by one. This supports a incremental development workflow of investigating intermediate results before coding the next cell.

You can also comment on your code using markdown formatted cells.

You later may export (main toolbar … > Export) your notebook e.g. as code script or as HTML with comments and outputs. You may then use standard print dialog and use the printer ("Als **PDF** speichern").

Key	
Cursor up/down	Navigate between cells
a / b	Insert a cell above or below
x / c / v	Cut / copy / paste a cell
z	Undo last cell operation
m / y / r	Changes the type of a cell to markdown, (Python) code, or raw (ignore it)
Enter / ESC	Toggle between edit mode and command mode
Ctrl-Enter Shift-Enter	Execute the cell (format markdown or execute the code). Shift-Enter additionally walks to the next cell.
TAB / Shift-TAB	Auto-completion / show signature in edit mode (IPython command – may take a second)
Shift-M	Merge cells
CTRL-Shift-minus	Split cell

Markdown

Markdown is a simple way to format text.

In JupyterLab, see Menu: Help > Markdown Reference for a cheat sheet and a tutorial.

You might also use LATEX to type mathematical formulas

```
A math formula using LaTeX: $a^2 + \frac{1}{b}$
```

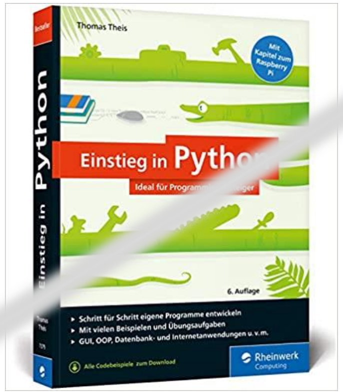
A math formula using LaTeX: $a^2 + \frac{1}{b}$

Type:	To get this output:
`*Italic*`	*Italic*
`**Bold**`	**Bold**
`# Heading 1`	Heading 1
`## Heading 2`	Heading 2
`* List` `* List` `* List`	• List • List • List
`---`	
`` `Inline code` with backticks ```	`Inline code` with backticks

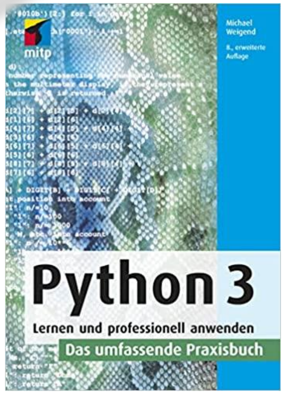
Website and Books

A good starting point is the homepage of Python itself
https://www.python.org/
There you find a tutorial https://docs.python.org/3/

Just for the sake of completeness:
We do also have many (> 140!) books in the library,
including (too) extensive introductions:



Thomas Theis

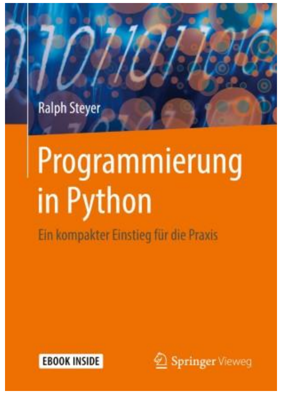


Michael Weigend

Online books

https://doi.org/10.1007/978-3-658-26133-7
(from within the HOST network or your VM)
Chapter 4 through 8 are just 40 pages.
You should at least go through this (read and try in your JupyterLab environment)!

https://doi.org/10.1007/978-3-658-28976-8
(from within the HOST network or your VM)



http://dx.doi.org/10.1007/978-3-658-20705-2
(from within the HOST network or your VM)

Sources for learning Python

Online courses

For those you have the time and like to go through an online video course, there are free courses on

- New openSAP course for beginners
  https://open.sap.com/courses/python1
- Udemy
- datacamp
- coursera
- MIT open courseware
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/

Kaggle

https://www.kaggle.com/

is a community for data science.

You will find datasets (!) together with code, discussions, courses, competitions,..

You can run your Notebooks there.

Python basics

Here comes a list of things, you need to understand at least to proceed:

- Basic syntax: indentation 4 blanks
- `print()`
- Writing string literals as
  raw string: `r'Backslash \n is printed here'`
  formatted string: `f'value of Variable {a}'`
- Data types `int, str`
- Data structures: list `[…]`, tupel `(…)`, dictionary `{…}`
- Defining a function with `def`
- Control flow: `if, while, for`
- `True, False, None`
- `assert 1 + 1 == 3, "1+1 should be 3"`
- `1_000_000` is the same as `1000000`

Helpful might be to know:

- `type(variable)` shows the data type of the variable
- `isinstance(df, pd.core.frame.DataFrame)` checks the type of the variable
- `dir(variable)` shows all attributes of the variable
- `help(variable)` shows documentation of the object

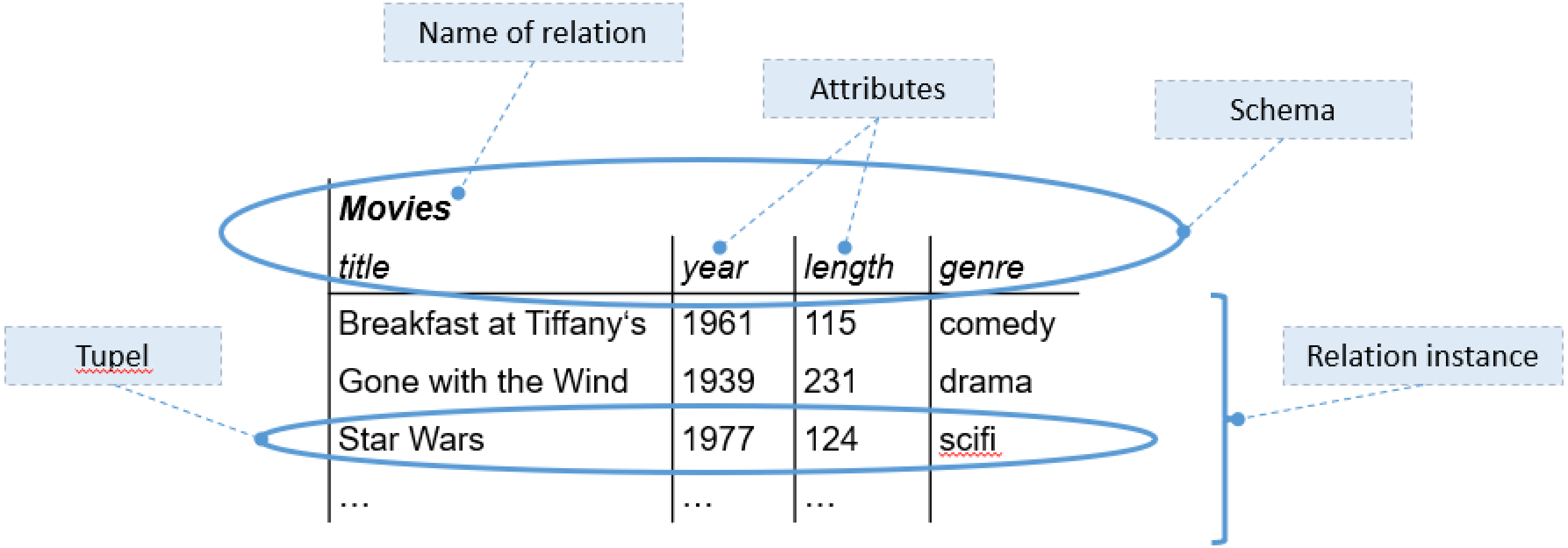
IPython commands:

- `variable?` gives Information about the variable
  `variable??` about the implementation, too
- `%xmode minimal` in JupyterLab (IPython) notebook reduces length of error messages
- `%timeit command` measures the time needed
- Output not just the result of the last command in a cell

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Schema

The Python pandas basic data structures DataFrame, Series and Index



pandas DataFrame

A **DataFrame** d with n rows and m columns is a two-dimensional array $d[i, j]$ for

$$i \in \{0, \dots, n-1\} = I_0,$$
$$j \in \{0, .., m-1\} = I_1,$$

plus an Index `d.index` to access rows

and an Index `d.columns` to access columns.

The data type of every column is uniformly defined.

Technically, a (row) **Index** is an array of labels $d.\mathrm{index}[i]$ for $i \in I_0$.

The labels do <u>not</u> have to be <u>unique</u>!

When no Index is specified, then $d.\mathrm{index}[i] = i$.

For a label l, we want to use the Index to access rows with $d.\mathrm{index}[i] = l$.

A **Series** is a one-dimensional DataFrame.

A DataFrame is comparable to a **DB relation**, but:

- <u>Ordering of rows</u> is fixed and relevant.
- Although it seems that a DataFrame is <u>symmetric with respect to rows and columns</u> in many respects, this is not 100% true: the data type is defined for columns not rows, and the naïve access is using columns / looping over rows. Although with `axis=1` many commands work equally on columns as they do on rows, for simplicity we will prefer sticking to a "relational" use of the DataFrame.
- RDMBS-<u>Index</u> columns belong to the relation, pandas-Index labels are not part of the data array $d[i, j]$. But they serve the same purpose: accessing rows efficiently.
  There is exactly one pandas (row-) Index at a time.
  The pandas Index for columns corresponds to the relation schema. Column labels do not have to be unique.

Constructor to create a DataFrame…

…from a Python Dictionary of Lists

```python
import pandas as pd
```

```python
data = {"year": [2000, 2001, 2002, 2000],
        "state": ['MV', 'MV', 'Saxen', 'Bayern'],
        "cases": [0, 1, 55, 33]}
df = pd.DataFrame(data)
```

```python
df
```

	year	state	cases
0	2000	MV	0
1	2001	MV	1
2	2002	Saxen	55
3	2000	Bayern	33

```sql
CREATE TABLE df AS SELECT …
SELECT * FROM df;
```

SQL

…from a Python List of Lists

```python
data = [[2000, 'MV', 0],
        [2001, 'MV', 1],
        [2002, 'Saxen', 55],
        [2001, 'Bayern', 33]]
pd.DataFrame(data)
```

	0	1	2
0	2000	MV	0
1	2001	MV	1
2	2002	Saxen	55
3	2001	Bayern	33

No Indexes defined

Explicitly define the Indexes for rows and columns:

```python
pd.DataFrame(data,
             columns=["year", "state", "cases"],
             index=["start", "one", "two", "tre"])
```

	year	state	cases
start	2000	MV	0
one	2001	MV	1
two	2002	Saxen	55
tre	2001	Bayern	33

Get schema information of a DataFrame: shape and data types

The shape is the number of rows and columns:

```
df.shape
```

```
(4, 3)
```

```
len(df)
```

```
4
```

The column data types are:

```
df.dtypes
```

```
year       int64
state     object
cases      int64
dtype: object
```

The (quite irrelevant) overall dtype of the DataFrame is `object`, since this is the only one covering `int64` and `object` columns.

```
DESCRIBE df;
SELECT COUNT(*) FROM df;
```

Oracle SQL

Available data types in pandas are:

Pandas dtype	Usage
`object`	Mixed data types, like Text (Python `str`), numeric, … – usually less efficient, since calculations are done on Python level
`Int64`	Integer numbers (not "int64")
`Float64`	Floating point numbers (not ("float64")
`boolean`	True / False values (not "bool")
`string`	This is not chosen by default ;-((`object` is preferred)
`datetime64[ns]`	Date and time with unit nanoseconds
`timedelta[ns]`	Differences between two datetimes
`category`	Enumerated list of values of some dtype

Get schema information of a DataFrame: Indexes

```
df.columns
```

```
Index(['year', 'state', 'cases'], dtype='object')
```

```
df.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

The RangeIndex is for the case $\mathrm{df.index}[i] = i$

```
[*pd.RangeIndex(start=0, stop=4, step=1, name='id')]
```

```
[0, 1, 2, 3]
```

The Indexes have a `name` attribute

```
df.columns.name = 'schema'
```

```
df.columns
```

```
Index(['year', 'state', 'cases'], dtype='object', name='schema')
```

```
df.index.name = 'id'
```

```
df.index
```

```
RangeIndex(start=0, stop=4, step=1, name='id')
```

A nice overview is given by

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   year    4 non-null      int64
 1   state   4 non-null      object
 2   cases   4 non-null      int64
dtypes: int64(2), object(1)
memory usage: 224.0+ bytes
```

```
DESCRIBE df;
SELECT COUNT(*) FROM df;
SELECT COUNT(*) FROM df WHERE year IS NOT NULL;
SELECT COUNT(*) FROM df WHERE state IS NOT NULL;
SELECT COUNT(*) FROM df WHERE cases IS NOT NULL;
```

Oracle SQL

Modify the dtype

`convert_dtypes()` tries to use the new datatypes:

```
data = {"year": [2000, 2001, 2002, 2000],
        "state": ['MV', 'MV', 'Saxen', 'Bayern'],
        "cases": [0, 1, 55, 33]}
df = pd.DataFrame(data).convert_dtypes()
```

```
df.dtypes
```

```
year      Int64
state    string
cases     Int64
dtype: object
```

Alternatively, we can define the desired dtypes in a Dictionary and apply them (we could do this already when reading data from CSV):

```
mydtypes = {"year": "string", "state": "object", "cases": "float64"}
df.astype(mydtypes).dtypes
```

Float64

```
year      string
state     object
cases    float64
dtype: object
```

Float64

Or use a helper function:

```
pd.to_numeric(df["year"])
```

```
0    2000
1    2001
2    2002
3    2000
Name: year, dtype: int64
```

```
pd.to_datetime(df["year"])
```

```
0   2000-01-01
1   2001-01-01
2   2002-01-01
3   2000-01-01
Name: year, dtype: datetime64[ns]
```

If we want to have this persistent, we have to overwrite the DataFrame or a column of it:

```
df.dtypes
```

```
year      Int64
state    string
cases     Int64
dtype: object
```

```
df = df.astype(mydtypes)
```

```
df.dtypes
```

```
year      string
state     object
cases    float64
dtype: object
```

Float64

```
df["state"] = df["state"].astype("string")
```

```
df.dtypes
```

```
year      string
state     string
cases    float64
dtype: object
```

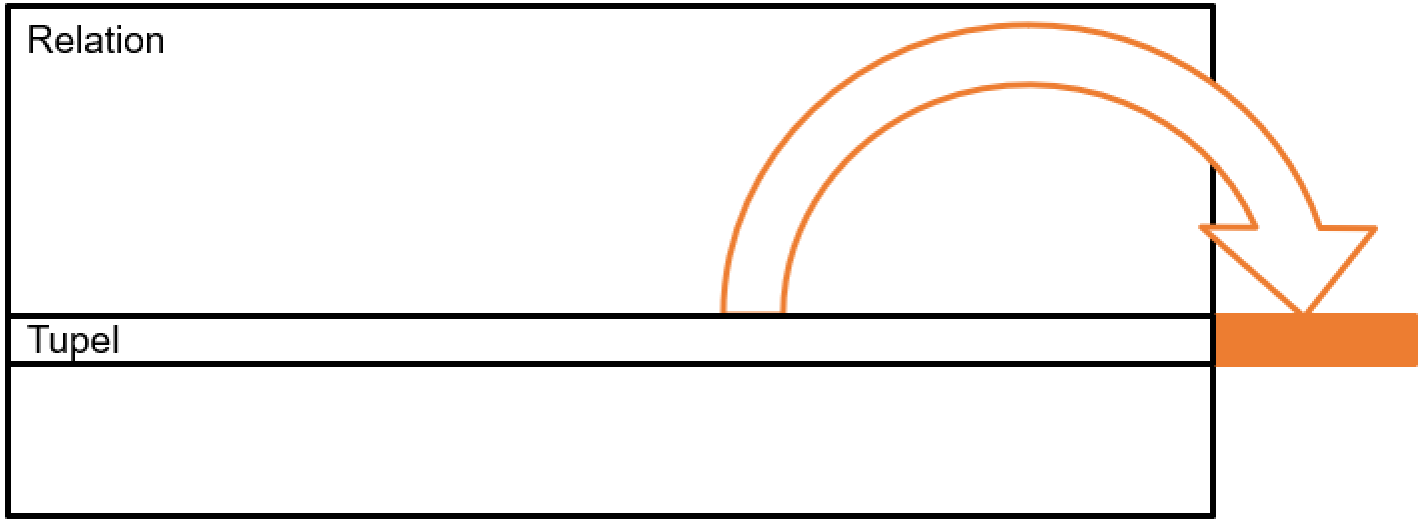
Float64

```
ALTER TABLE df
MODIFY state VARCHAR2(100);
```

SQL

Projection

Working on columns and computing new columns



Projection (one column)

$\pi_{\text{year}}(\text{df})$:

```
df["year"]
```

```
0    2000
1    2001
2    2002
3    2000
```

```
SELECT "year" FROM df;
```

SQL

Please avoid writing `df.year` (can be confused with a method "year"; works only with Python identifiers – not with blanks in the name e.g.).

A note on **quotations marks**:

In Python you can use `"` and `'` interchangeably.

We will use this rule similar to SQL code:

- For string literals, use single quotes.
- For identifiers, use double quotes.

```
df["state"] = 'FarAway'
df
```

	year	state	cases
0	2000	FarAway	0.0
1	2001	FarAway	1.0
2	2002	FarAway	55.0
3	2000	FarAway	33.0

```
UPDATE df SET "year" = 'FarAway';
```

SQL

Note that this one-dimensional result is a Series:

```
type(df["year"])
```

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

We can construct a DataFrame from one or more Series, or, from many other objects:

```
years = df["year"]
pd.DataFrame(years)
```

	year
0	2000
1	2001
2	2002
3	2000

```
states = df["state"]
pd.DataFrame({"year":years, "state":states})
```

	year	state
0	2000	MV
1	2001	MV
2	2002	Saxen
3	2000	Bayern

This only works,
because both Series (`years` and `states`)
have the same index `[0, 1, 2, 3]`.

Projection (multiple columns)

$\pi_{\text{cases, state, cases, year}}(\text{df})$:

```
SELECT "year", "state", "year"
FROM df;
```

SQL

```
dup = df[["cases","state","cases","year"]]
dup
```

	cases	state	cases	year
0	0	MV	0	2000
1	1	MV	1	2001
2	55	Saxen	55	2002
3	33	Bayern	33	2000

This way, we can also reorder the columns of a DataFrame.

If the column index is not unique, then the result is not a Series, but will contain all matching columns:

```
dup["cases"]
```

	cases	cases
0	0	0
1	1	1
2	55	55
3	33	33

Extended projection

Unlike SQL, this only works for one calculated column at a time.

```
SELECT "death" / "cases" * 100
FROM df;
```

SQL

```
df["death"] / df["cases"] * 100
```

```
0          NaN
1     0.000000
2    12.727273
3    12.121212
dtype: float64
```

What happens?

Two Series are aligned according to the index and division is computed for each element.

We can use the result to create a **new column** in the DataFrame (or to overwrite an existing column):

```
df["mortality"] = df["death"] / df["cases"] * 100
```

```
df
```

	year	state	cases	death	mortality
0	2000	MV	0	0	NaN
1	2001	MV	1	0	0.000000
2	2002	Saxen	55	7	12.727273
3	2000	Bayern	33	4	12.121212

For more complex calculations, we need the `apply()` method (see section on grouping/aggregatoin later in this chapter).

Extended projection

Alternatively, use `assign()`:

```
df.assign(mortality = df["death"] / df["cases"] * 100)
```

	year	state	cases	death	mortality
0	2000	MV	0	0	NaN
1	2001	MV	1	0	0.000000
2	2002	Saxen	55	7	12.727273
3	2000	Bayern	33	4	12.121212

Remember that the original DataFrame `df` is not modified.

Con: Only simple identifiers are possible.

Pro: It can be chained with other transformations of a DataFrame. Multiple calculated columns are possible in one command (like in SQL).

Rename a column

```
df.rename(columns={"mortality": "m", "year": "y"})
```

	y	state	cases	death	m
0	2000	MV	0	0	NaN
1	2001	MV	1	0	0.000000
2	2002	Saxen	55	7	12.727273
3	2000	Bayern	33	4	12.121212

```
SELECT "year" AS "y",
    state,
    cases,
    death,
    "mortality" AS "m"
FROM df;
```
SQL

Drop a column

```
ALTER TABLE df DROP COLUMN mortality;
ALTER TABLE df DROP COLUMN year;
```
SQL

This can be achieved using the Python `del` command, the pandas `pop()` or `drop()` methods.

```
df.drop(columns=["mortality", "year"], inplace=True)
```

```
df
```

	state	cases	death
0	MV	0	0
1	MV	1	0
2	Saxen	55	7
3	Bayern	33	4

`inplace=True` will do it in the original DataFrame.

`inplace=False` (the default) will return the modified DataFrame. This can be used to chain commands.

Alternatively, select the columns you do <u>not</u> want to drop:

```
df = df[["state", "cases", "death"]]
```

```
df
```

	state	cases	death
0	MV	0	0
1	MV	1	0
2	Saxen	55	7
3	Bayern	33	4

Interlude

Some remarks not fitting elsewhere

Shallow vs. deep copy (SettingWithCopyWarning)

Achtung shallow oder deep copy

Bei einer shallow copy warnt ggf. IPython, wenn man anschließend eine Zuweisu

Diese wird dennoch ausgeführt und verändert dann auch df.

```
shallow = df[["state","cases"]]
```

```
shallow["cases"] += 100
```

```
<ipython-input-77-9e91dec35f3c>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/panda
  shallow["cases"] += 100
```

```
df
```

	state	cases	death
0	MV	100	0
1	MV	101	0
2	Saxen	155	7
3	Bayern	133	4

Simple assignment is doing only a shallow copy (= reference = a view).

So sometimes, the outcome (view or copy) is unpredictable and that is the reason for the `SettingWithCopyWarning`.

https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#why-does-assignment-fail-when-using-chained-indexing

Normally, we don't have to worry too much about it. Usually, we will subsequently assign intermediate results to new variables and not tamper with "old" variables any more.

To avoid the warning,
do an explicit
deep copy earlier in your code.

```
deep = df[["state","cases"]].copy()
```

```
deep["cases"] += 2000
```

```
df
```

	state	cases	death
0	MV	100	0
1	MV	101	0
2	Saxen	155	7
3	Bayern	133	4

NEW: Always do a **Copy on Write**

https://phofl.github.io/cow-introduction.html

```
pd.set_option("mode.copy_on_write", True)
```

Other findings so far

- If you want to modify the original df, assign the result to the same Series / DataFrame again (`df = …`), or use `inplace=True`.
- Operations on Series / DataFrame usually work elementwise. First, the Series / DataFrame are aligned according to its indexes.

```
df*2
```

	state	cases	death
0	MVMV	0	0
1	MVMV	2	0
2	SaxenSaxen	110	14
3	BayernBayern	66	8

What is left to do

We need to finish the standard SQL topics:

- Operations on rows (selection, sorting)
- INSERT / UPDATE / DELETE
- NULLs
- Grouping and aggregation
- Joins

Then, we can do more fancy stuff like time series, …

You might hear a lot about NumPy

It is a library like pandas. pandas uses NumPy internally. It is optimized for mathematical vector operations. It has different data types. For simplicity, we are avoiding to know and use NumPy directly.

Fancy Python tools

PEP8 defines coding syntax, e.g. whitespace
https://pep8.org/#introduction

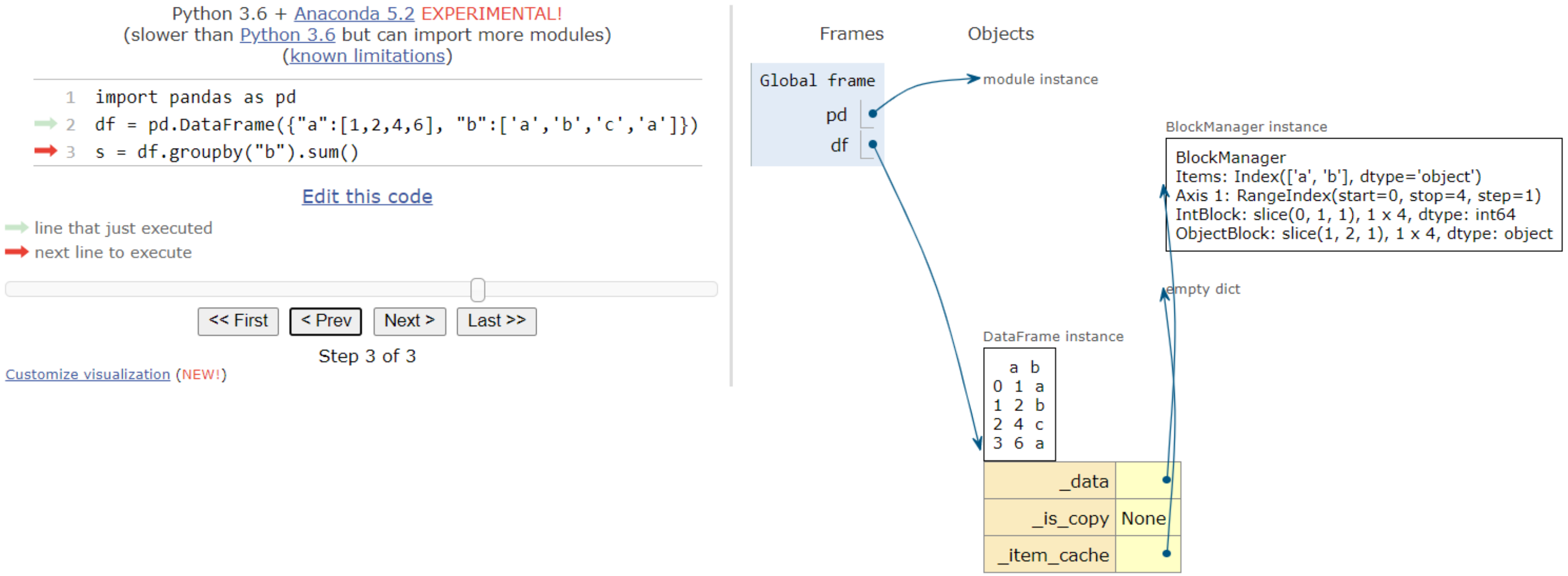
This syntax can be automatically checked.

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

PEP20 defines coding principles.

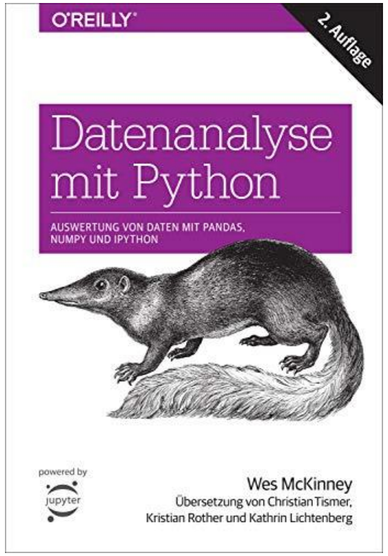
The http://www.pythontutor.com/ visualizes code. This helps to understand what's happening behind the scenes.

But it has some limitations (loading csv from pandas not possible), which may makes not so helpful. Rather suitable for small learning examples.

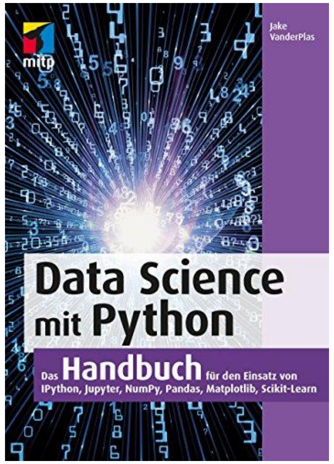


Literature

Wes McKinney is the founder of pandas and hence his book is a good reference.



There are more Python and Python pandas books in our library.



There are so many tutorials, videos and online courses: google what you need, but don't get too confused!

Besides these sources, the important reference is the **pandas home page** https://pandas.pydata.org/pandas-docs/stable/

When using a command, check out the API reference first.

You will find many suggestions in the net, but here you find the correct or good way of doing it. Take into account that the pandas package is still under development and avoid outdated syntax.

Read and write data

Pandas supports reading/writing data from/to various sources like: CSV, text, **MS Excel**, **JSON**, **XML**, **HTML webpages**, **SQL**, **SAS**,…

We focus on **CSV** now.

If you reading CSV manually (e.g. in Python) you come across many problems:

- header line?
- separator is a comma or a semicolon (as in Germany)?
- quoting of special characters with "" – quoting of quotes?
- number format, thousands separator
- Fields with multiline text (it is not sufficient to read a file line-by-line)

A help is the Python `csv`-package, but even better is the pandas `read_csv()` command. It is basically a worry-free package and has lots of options if needed.

```
pd.read_csv('GBI_Data.csv')
```

	OrderNumber	OrderItem	YEAR	MONTH	Date	Customer	CustDescr	City	SalesOr
0	100001	10	2007	1	2007-01-01	17000	Cruiser Bikes	Hannover	DN0
1	100001	20	2007	1	2007-01-01	17000	Cruiser Bikes	Hannover	DN0
2	100001	30	2007	1	2007-01-01	17000	Cruiser Bikes	Hannover	DN0

Exercise: improve the command to take care of data types and remove superfluous columns. Save the modified data as CSV and as MS Excel.

We even can read CSV from a URL:

```
url = ("https://raw.github.com/pandas-dev"
       "/pandas/master/pandas/tests/io/data/csv/tips.csv")
tips = pd.read_csv(url)
tips.head(3)
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3

From a database

This case depends on the database brand. You need a specific driver to connect to the DB. In the case of Oracle, you would need to install cx_oracle first.

```
import cx_Oracle
```

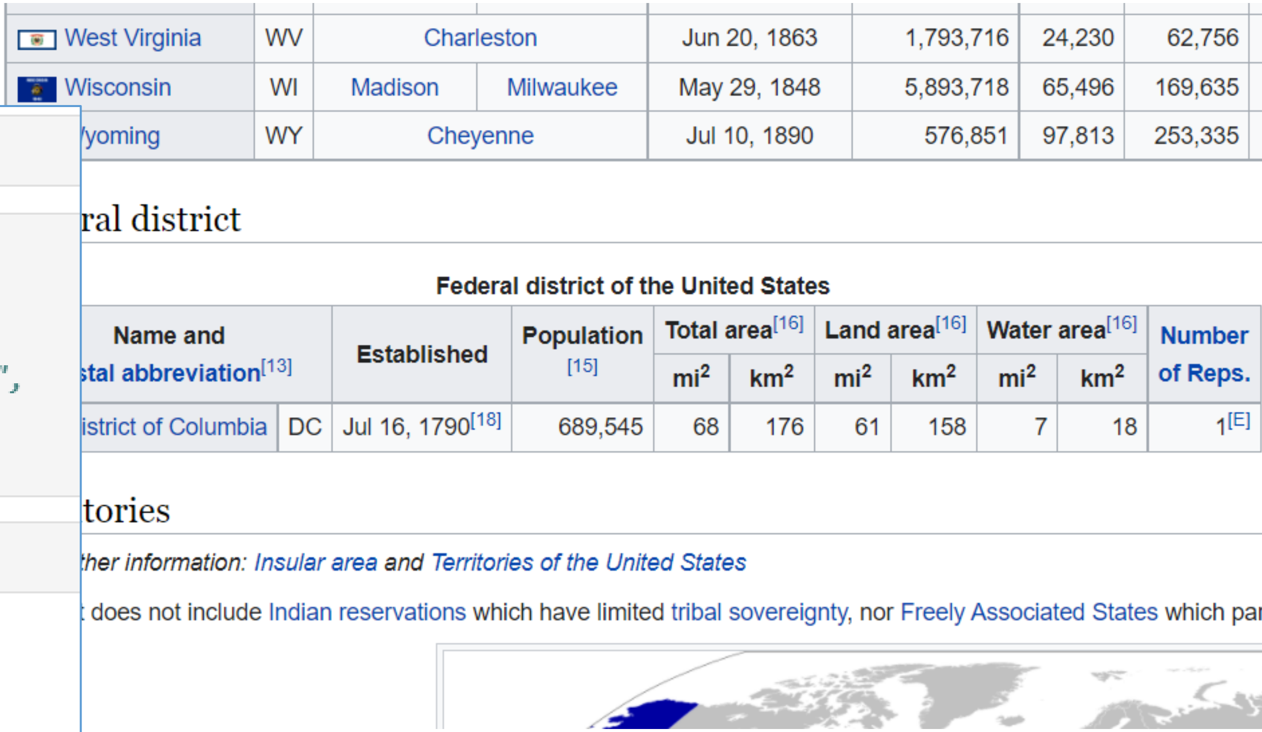
```
# Connect to the DB
DBDsn = "fbwdb/xepdb1"
DBUser = "DB00"
with open("DB00Passwort.txt", "r") as f:  #contains just one word\n",
    DBPassword = f.read()
DBConnection = cx_Oracle.connect(DBUser, DBPassword, DBDsn)
```

```
pd.read_sql(con=DBConnection, sql='SELECT * FROM corona')
```

	DATUM	STADT	LAND	NEUEFAELLE	EMAIL	VERSTORBENE
0	2019-11-12	Stralsund	M-V	3	HST@me.de	1
1	2020-11-13	Stralsund	M-V	5	HST@me.de	0
2	2020-11-14	Stralsund	M-V	-42	HST@me.de	1

From a webpage

See exercises!



West Virginia	WV	Charleston		Jun 20, 1863	1,793,716	24,230	62,756
Wisconsin	WI	Madison	Milwaukee	May 29, 1848	5,893,718	65,496	169,635
yoming	WY	Cheyenne		Jul 10, 1890	576,851	97,813	253,335

ral district

Federal district of the United States

Name and stal abbreviation[13]		Established	Population [15]	Total area[16]		Land area[16]		Water area[16]		Number of Reps.
				mi²	km²	mi²	km²	mi²	km²	
istrict of Columbia	DC	Jul 16, 1790[18]	689,545	68	176	61	158	7	18	1[E]

tories

ther information: Insular area and Territories of the United States

does not include Indian reservations which have limited tribal sovereignty, nor Freely Associated States which pa

Instead of CSV

CSV is a well-known format, but not very efficient.

To save space (and thus time), use a modern format like these:

```
large_df.to_parquet('output.parquet')
large_df.to_feather('output.feather')
large_df.to_pickle('output.pickle')
```

In addition, these file formats also retain the data type of each column.

Print a df or a Series

When we display the contents of a df or Series in the JupyterLab notebook, the output is truncated when too large.

```
url = ("https://raw.github.com/pandas-dev"
       "/pandas/master/pandas/tests/io/data/csv/tips.csv")
tips = pd.read_csv(url)
tips
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

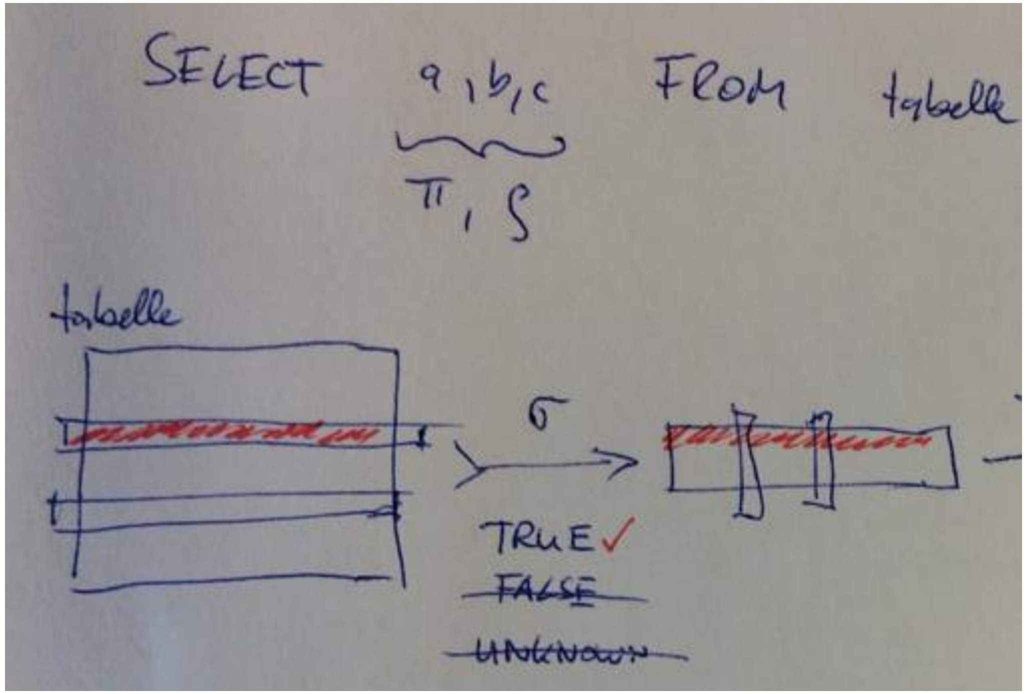
If we want to overcome this, here is the trick:

```
print(tips.to_string())
```

```
    total_bill   tip     sex smoker   day    time  size
0        16.99  1.01  Female     No   Sun  Dinner     2
1        10.34  1.66    Male     No   Sun  Dinner     3
2        21.01  3.50    Male     No   Sun  Dinner     3
3        23.68  3.31    Male     No   Sun  Dinner     2
4        24.59  3.61  Female     No   Sun  Dinner     4
5        25.29  4.71    Male     No   Sun  Dinner     4
6         8.77  2.00    Male     No   Sun  Dinner     2
7        26.88  3.12    Male     No   Sun  Dinner     4
8        15.04  1.96    Male     No   Sun  Dinner     2
9        14.78  3.23    Male     No   Sun  Dinner     2
10       10.27  1.71    Male     No   Sun  Dinner     2
11       35.26  5.00  Female     No   Sun  Dinner     4
12       15.42  1.57    Male     No   Sun  Dinner     2
13       18.43  3.00    Male     No   Sun  Dinner     4
14       14.83  3.02  Female     No   Sun  Dinner     2
15       21.58  3.92    Male     No   Sun  Dinner     2
16       10.33  1.67  Female     No   Sun  Dinner     3
17       16.29  3.71    Male     No   Sun  Dinner     3
18       16.97  3.50  Female     No   Sun  Dinner     3
19       20.65  3.35    Male     No   Sat  Dinner     3
20       17.92  4.08    Male     No   Sat  Dinner     2
21       20.29  2.75  Female     No   Sat  Dinner     2
22       15.77  2.23  Female     No   Sat  Dinner     2
23       39.42  7.58    Male     No   Sat  Dinner     4
24       19.82  3.18    Male     No   Sat  Dinner     2
25       17.81  2.34    Male     No   Sat  Dinner     4
26       13.37  2.00    Male     No   Sat  Dinner     2
```

Selection

Selection of rows



Index

Define an index (yet a MultiIndex with multiple columns)

```
# gbi = pd.read_csv('GBI_Data.csv')
gbi.set_index(["Date", "Customer", "Product"], inplace=True)
```

```
assert(gbi.index.is_unique)
```

```
AssertionError
```

```
gbi.head()
```

			OrderNumber	OrderItem	YEAR	MONTH	CustDescr	City	SalesOrg
Date	**Customer**	**Product**							
2007-01-01	**17000**	**PRTR1000**	100001	10	2007	1	Cruiser Bikes	Hannover	DN00
		DXTR1000	100001	20	2007	1	Cruiser Bikes	Hannover	DN00
		DXRD2000	100001	30	2007	1	Cruiser Bikes	Hannover	DN00
		ORWN1000	100001	40	2007	1	Cruiser Bikes	Hannover	DN00
2007-01-03	**15000**	**PRTR1000**	100002	10	2007	1	Bavaria Bikes	München	DS00

Remove the index (replace it by the RangeIndex)

```
gbi.reset_index(inplace=True)
```

```
gbi.head()
```

	Date	Customer	Product	OrderNumber	OrderItem	YEAR	MONTH	CustDescr	Cit
0	2007-01-01	17000	PRTR1000	100001	10	2007	1	Cruiser Bikes	Hannove
1	2007-01-01	17000	DXTR1000	100001	20	2007	1	Cruiser Bikes	Hannove
2	2007-01-01	17000	DXRD2000	100001	30	2007	1	Cruiser Bikes	Hannove
3	2007-01-01	17000	ORWN1000	100001	40	2007	1	Cruiser Bikes	Hannove

Selection using the index (labels)

Remarks about the index:

- There is exactly one index at a time
  (some columns, or, an artificial RangeIndex)
- The purpose of the index is not having faster access as it is true for RDBMS. The purpose is basically to explicitly access certain rows, or, for aligning data (for joining along the index).
- You don't have to use indexes at all, or, redefine a suitable index for each and every different access. But when there is a "natural" index, it might make sense to use it. Operations in the sequel might be easier then.
- When joining two DataFrames via non-index columns, the information in the index "columns" is "lost" (not contained in the result). You can avoid this be using `reset_index()` before the join.

Access of rows via the index using `loc[]`:

```
gbi.loc['2007-01-01']
```

			OrderNumber	OrderItem	CustDescr	City	SalesOrg	Country
Date	**Customer**	**Product**						
2007-01-01	**17000**	**PRTR1000**	100001	10	Cruiser Bikes	Hannover	DN00	DE
		DXTR1000	100001	20	Cruiser Bikes	Hannover	DN00	DE
		DXRD2000	100001	30	Cruiser Bikes	Hannover	DN00	DE
		ORWN1000	100001	40	Cruiser Bikes	Hannover	DN00	DE

```
SELECT * FROM gbi
WHERE "Date" = '2007-01-01';   -- Date is a string
```

SQL

Some `loc[]` examples using the index

Slicing is possible, i.e. defining an interval instead of a single value. Syntax for a slice is:

`a:b`	means all values in $[a, b]$
`:b`	means all values in $\leq b$
`a:`	means all values in $\geq$ a
`:`	means all values (no restriction)

```
gbi.loc['2007-01-01':'2007-01-29']
```

```
SELECT * FROM gbi
WHERE "Date" BETWEEN '2007-01-01' AND '2007-01-29';
```

SQL

Caution 1: When the index is not sorted you may get a

```
PerformanceWarning: indexing past lexsort depth may impact performance.
```

```
gbi.index.is_monotonic_increasing
```

```
False
```

remedy →

```
gbi.sort_index()
```

Caution 2: The <u>index</u> must be <u>sorted</u> to allow slicing:

```
gbi = gbi.sort_values(by="Product")
```

```
gbi.loc['2007-01-01':'2007-01-29']
```

```
UnsortedIndexError: 'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

For a **MultiIndex**, you can specify all columns:

```
gbi.loc['2007-01-01', 17000, ['PRTR1000','DXTR1000']]
```

```
SELECT * FROM gbi
WHERE "Date" = '2007-01-01'
AND "Customer" = 17000
AND "Product IN ('PRTR1000','DXTR1000');
```

SQL

and use slicing-wildcard at every position:

```
gbi.loc[:, 17000, :]
```

```
SELECT * FROM gbi
WHERE "Customer" = 17000;
```

SQL

Caution: Again, slicing depends on sorting. Hence, more complex slicing scenarios may fail.

A MultiIndex is not good for slicing or yet more complex logical conditions.

`loc[]` horizontally

Besides selection of rows, the method `loc[]` also allows to "select" columns (**projection**).

`df.loc[`*row_indexer*`,` *column_indexer*`]`

Hence, the following two expressions are identical:

```
gbi.loc[:, "City"] == gbi["City"]
```

<u>Warning</u>: By convention, the syntax `df[df.year>2000]` (masking) or `df[1:3]` (slicing) is interpreted to work on rows, not columns. This may be convenient sometimes, but is very confusing. We advise not to use this ("syntactical sugar" as Wes McKinney names it).

We would like to advise to stick with the `df[…]` notation for <u>projections</u> of columns and `df.loc[…]` for <u>selections</u> of rows using the index.

We can do both at the same time

```
gbi.loc['2007-01-01']["City"]
```

```
Date        Customer  Product
2007-01-01  17000     PRTR1000    Hannover
```

But we cannot use this on the left side of an assignment

```
gbi.loc['2007-01-01']["City"] = 'Atlantis'
```

```
<ipython-input-19-8ba73951bc2d>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

In this case, we need `.loc[]` with the *column_indexer*

```
gbi.loc['2007-01-01', "City"] = 'Atlantis'
```

```
UPDATE gbi
SET "City" = 'Atlantis'
WHERE "Date" = '2007-01-01';
```

SQL

Problems with `loc[row_indexer, col_indexer]` for MultiIndex

For a MultiIndex having both, row- and column-indexer gives an error (two many commas):

```
gbi.loc['2007-01-01', 17000, ['PRTR1000','DXTR1000'], ["City", "Country"]]
```

```
IndexError: list index out of range
```

The elements of a MultiIndex are Tuples: so we better write it with parenthesis:

```
gbi.loc[('2007-01-01', 17000, ['PRTR1000','DXTR1000']), ["City", "Country"]]
```

Date	Customer	Product	City	Country
2007-01-01	17000	PRTR1000	Hannover	DE
		DXTR1000	Hannover	DE

This does not work in combination with slicing:

```
gbi.loc[('2007-01-01', 17000:18000, ['PRTR1000','DXTR1000']), ["City", "Country"]]
```

```
  File "<ipython-input-51-99581a08af90>", line 1
    gbi.loc[('2007-01-01', 17000:18000, ['PRTR1000','DXTR1000']), ["City", "Country"]]
                                ^
SyntaxError: invalid syntax
```

Solution for MultiIndex and slicing at the same time: use "`pd.IndexSlice`":

```
gbi.loc[pd.IndexSlice['2007-01-01', 17000:18000, ['PRTR1000','DXTR1000']], ["City", "Country"]]
```

Date	Customer	Product	City	Country
2007-01-01	17000	PRTR1000	Hannover	DE
		DXTR1000	Hannover	DE

Selection by position (`iloc[]`)

`iloc[]` accesses rows and columns not via the index labels like `loc[]`, but using <u>integer positions</u>.

Remember, our DataFrame is a two-dimensional array $d[i,j]$ for $i \in \{0,\ldots,n-1\} = I_0, j \in \{0,..,m-1\} = I_1$.

Then, `d.iloc[a:b, c:d]`
is the data set $d[i,j]$ for $i \in [a,b] \cap I_0, j \in [c,d] \cap I_1$
(extend it to special cases of slicing where not both bounds are given; the (row) index and columns (index) are sliced accordingly).

```
tips.iloc[:3]
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3

```sql
-- ordering or rows is immaterial for relations
```
SQL

Both `.iloc[]` and `.loc[]` also take <u>lists</u> of values:

```
tips.iloc[[5,2,3]]
```

	total_bill	tip	sex	smoker	day	time	size
5	25.29	4.71	Male	No	Sun	Dinner	4
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2

Vector arithmetic and alignment of DataFrame, Series and Scalars

1) First, for operations between objects, they are aligned according to the index and the columns (index)
2) Then, the operation is done element-wise.
3) Missing values in the result (due to alignment) are filled with `pd.NA` ("NULL" value).

Two Series:

$$\left\|\begin{array}{c} a\\ c\\ e\\ m \end{array}\begin{bmatrix} 7.3\\ -2\\ 1.1\\ 4 \end{bmatrix}\right\| + \left\|\begin{array}{c} a\\ d\\ e \end{array}\begin{bmatrix} 1\\ -2\\ 4 \end{bmatrix}\right\| = \left\|\begin{array}{c} a\\ c\\ d\\ e\\ m \end{array}\begin{bmatrix} 8.3\\ <\text{NA}>\\ <\text{NA}>\\ 5.1\\ <\text{NA}> \end{bmatrix}\right\|$$

Two DataFrames:

$$\left\|\begin{array}{c} \\ a\\ c\\ e\\ m \end{array}\begin{array}{c} \begin{array}{cc} V & W \end{array}\\ \begin{bmatrix} 7.3 & 7\\ -2 & 7\\ 1.1 & 7\\ 4 & 7 \end{bmatrix} \end{array}\right\| + \left\|\begin{array}{c} \\ a\\ d\\ e \end{array}\begin{array}{c} \begin{array}{cc} K & V \end{array}\\ \begin{bmatrix} 1 & 2\\ -2 & 5\\ 4 & 2 \end{bmatrix} \end{array}\right\| = \left\|\begin{array}{c} \\ a\\ c\\ d\\ e\\ m \end{array}\begin{array}{c} \begin{array}{ccc} K & V & W \end{array}\\ \begin{bmatrix} <\text{NA}> & 9.3 & <\text{NA}>\\ <\text{NA}> & <\text{NA}> & <\text{NA}>\\ <\text{NA}> & <\text{NA}> & <\text{NA}>\\ <\text{NA}> & 3.1 & <\text{NA}>\\ <\text{NA}> & <\text{NA}> & <\text{NA}> \end{bmatrix} \end{array}\right\|$$

DataFrame and Series is more complicated.

More interesting for us now is **DataFrame/Series and Scalar** (index and columns remain unchanged):

$$\left\|\begin{array}{c} m\\ n\\ o\\ p \end{array}\begin{bmatrix} 2.3 & 2\\ -2 & 2\\ 1.1 & 3\\ 4 & 3 \end{bmatrix}\right\| * 2 = \left\|\begin{array}{c} m\\ n\\ o\\ p \end{array}\begin{bmatrix} 4.6 & 4\\ -4 & 4\\ 2.2 & 6\\ 8 & 6 \end{bmatrix}\right\|$$

$$\left(\left\|\begin{array}{c} m\\ n\\ o\\ p \end{array}\begin{bmatrix} 2.3 & 2\\ -2 & 2\\ 1.1 & 3\\ 4 & 3 \end{bmatrix}\right\| < 0\right) = \left\|\begin{array}{c} m\\ n\\ o\\ p \end{array}\begin{bmatrix} \text{False} & \text{False}\\ \text{True} & \text{False}\\ \text{False} & \text{False}\\ \text{False} & \text{False} \end{bmatrix}\right\|$$

$$\left(\left\|\begin{array}{c} m\\ n\\ o\\ p \end{array}\begin{bmatrix} 3\\ -2\\ 3\\ -3 \end{bmatrix}\right\| == 3\right) = \left\|\begin{array}{c} m\\ n\\ o\\ p \end{array}\begin{bmatrix} \text{True}\\ \text{False}\\ \text{True}\\ \text{False} \end{bmatrix}\right\|$$

Once again two Series, now with a logical operator `&` (and)

$$\left\|\begin{array}{c} a\\ c\\ e\\ m \end{array}\begin{bmatrix} \text{True}\\ \text{True}\\ \text{True}\\ \text{False} \end{bmatrix}\right\| \,\&\, \left\|\begin{array}{c} a\\ c\\ e\\ m \end{array}\begin{bmatrix} \text{True}\\ \text{False}\\ <\text{NA}>\\ \text{False} \end{bmatrix}\right\| = \left\|\begin{array}{c} a\\ c\\ e\\ m \end{array}\begin{bmatrix} \text{True}\\ \text{False}\\ <\text{NA}>\\ \text{False} \end{bmatrix}\right\|$$

Selection with `loc[]` for non-index columns

Doing a logical operation with a Series defines a True/False/<NA> vector:

```
tips["tip"] > 7
```

```
0       False
1       False
2       False
3       False
4       False
        ...
```

This vector can be used to filter rows in the DataFrame:

```
tips.loc[tips["tip"] > 7]
```

	total_bill	tip	sex	smoker	day	time	size
23	39.42	7.58	Male	No	Sat	Dinner	4
170	50.81	10.00	Male	Yes	Sat	Dinner	3
212	48.33	9.00	Male	No	Sat	Dinner	4

```
SELECT * FROM tips
WHERE tip > 7;
```
SQL

More complicated Boolean expressions using `&` (and), `|` (or), `^` (exclusive-or), `~` (not) and `<`, `>`, `!=`, `==`, `isin()`.
Caution: do not use the Python keywords `and`, `or`, `not`, because we need to work on vectors, not scalars.

```
(tips["tip"] > 7) & ~(tips["smoker"] == 'Yes')
```

```
0       False
1       False
2       False
3       False
4       False
        ...
239     False
240     False
241     False
242     False
243     False
Length: 244, dtype: bool
```

```
SELECT * FROM tips
WHERE tip > 7
AND NOT smoker = 'Yes';
```
SQL

```
tips.loc[(tips["tip"] > 7) & ~(tips["smoker"] == 'Yes')]
```

	total_bill	tip	sex	smoker	day	time	size
23	39.42	7.58	Male	No	Sat	Dinner	4
212	48.33	9.00	Male	No	Sat	Dinner	4

Two more simple operations on rows

`head() / tail() / sample()`

Return the top n rows – for the equivalent in RDBMS we would need to use a suitable sorting.

```
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

```
tips.head(2)
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3

```
SELECT * FROM tips
WHERE ROWNUM < 6
ORDER BY ???;
```

Oracle SQL

```
SELECT * FROM tips
FETCH NEXT 5 ROWS ONLY
ORDER BY ???;
```

Oracle SQL

```
SELECT TOP 5 *
FROM tips
ORDER BY ???;
```

SAP HANA SQL

Sorting

…using the index,

```
tips.sort_index(inplace=True)
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

or using some columns (also works for the index)

```
tips.sort_values(by=["total_bill", "sex"], ascending=[True, False], inplace=True)
tips
```

	total_bill	tip	sex	smoker	day	time	size
67	3.07	1.00	Female	Yes	Sat	Dinner	1
92	5.75	1.00	Female	Yes	Fri	Dinner	2
172	7.25	5.15	Male	Yes	Sun	Dinner	2
111	7.25	1.00	Female	No	Sat	Dinner	1
149	7.51	2.00	Male	No	Thur	Lunch	2
...	...	...	...	...	...	...	...

```
SELECT * FROM tips
ORDER BY "total_bill", "sex" DESC;
```

SAP HANA SQL

pd.NA

Working with NULL values

SQL	evaluates to
`1 + 2`	`3`
`1 + NULL`	`NULL`
`3 * NULL`	`NULL`
`1 > NULL`	`UNKNOWN`
`1 = 1`	`TRUE`
`42 = NULL`	`UNKNOWN`
`NULL = NULL`	`UNKNOWN`
`NULL IS NULL`	`TRUE`
`(1=1) AND (1=1)`	`TRUE`
`(1=1) AND (1=NULL)`	`UNKNOWN`

NOT(A)

A	¬A
F	T
U	U
T	F

AND(A, B)

A ∧ B		B		
		F	U	T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

OR(A, B)

A ∨ B		B		
		F	U	T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

The truth about `pd.NA`

There is a bunch of values, which might serve as a database NULL equivalent:

- The Python `None` value
- In NumPy there is the notion of a not-a-number `np.NaN`. This is a float datatype and cannot be used across all datatypes. It was used in pandas in the past.
- Pandas introduced a new value `pd.NA`

We recommend using **`pd.NA`** only (displayed as "**<NA>**" / "**NaN**" / "**NA**").

```python
import numpy as np
```

```python
df = pd.DataFrame({"col": [1, "text", None, pd.NA, np.NaN]})
df
```

	col
0	1
1	text
2	None
3	<NA>
4	NaN

```python
df.isna()
```

	col
0	False
1	False
2	True
3	True
4	True

pd.NA is not available for all datatypes in pandas yet – this subject to change

```python
None == None
```

```
True
```

```python
not(np.NaN == np.NaN)
```

```
True
```

In contrast, `pd.NA` implements the expected logic:

```python
pd.NA == pd.NA
```

```
<NA>
```

```python
pd.NA is pd.NA
```

```
True
```

```python
1 + pd.NA
```

```
<NA>
```

```python
1 > pd.NA
```

```
<NA>
```

```python
pd.NA & True
```

```
<NA>
```

```python
pd.NA | True
```

```
True
```

Problems with `pd.NA`

Python `bool` only has the values `True` / `False`

```
s = pd.Series([1,2,pd.NA])
s == 2
```

```
0    False
1     True
2    False
dtype: bool
```

```
s == pd.NA
```

```
0    False
1    False
2    False
dtype: bool
```

Even worse:

```
if(pd.NA < 13):
    print('kleiner')
else:
    print('groß')
```

```
TypeError: boolean value of NA is ambiguous
```

Solution: explicitly check for Null-values using `pd.isna()`

Using `pd.NA`

We can normalize all values to be `pd.NA`

```
df.fillna(value=pd.NA)
```

	col
0	1
1	text
2	<NA>
3	<NA>
4	<NA>

We can fill missing data, or, drop rows with missing data:

```
df.fillna(value=0)
```

	col
0	1
1	text
2	0
3	0
4	0

```
df.dropna()
```

	col
0	1
1	text

Converting the data type to numeric may fail,

```
df = pd.DataFrame({"year": ["2020", 2021, None, "hugo"]})
df
```

	year
0	2020
1	2021
2	None
3	hugo

```
pd.to_numeric(df["year"])
```

```
ValueError: Unable to parse string "hugo"
```

but can be marked by `pd.NA` values:

```
pd.to_numeric(df["year"], errors="coerce").convert_dtypes()
```

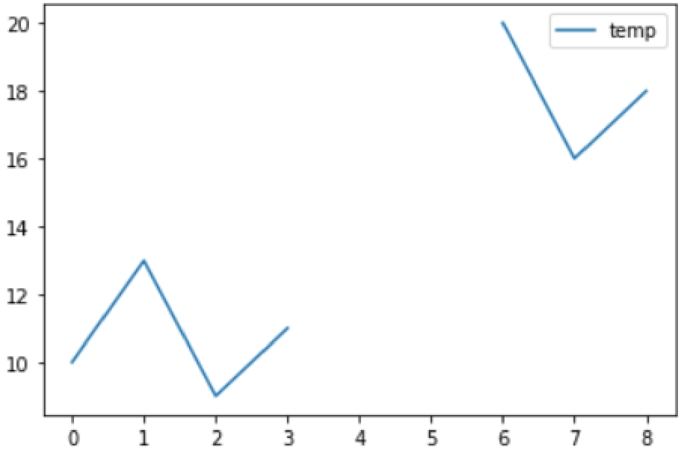
```
0    2020
1    2021
2    <NA>
3    <NA>
Name: year, dtype: Int64
```

Interpolate missing values

```
df = pd.DataFrame({"temp": [10, 13, 9, 11, None, None, 20, 16, 18]}, dtype="float64")
```

```
df.plot()
```

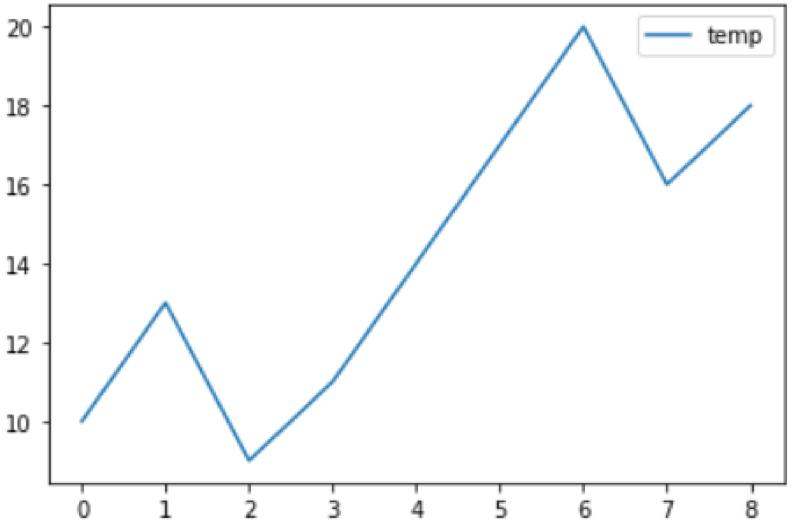
<AxesSubplot:>



Missing values can be linear interpolated:

```
df.interpolate().plot()
```

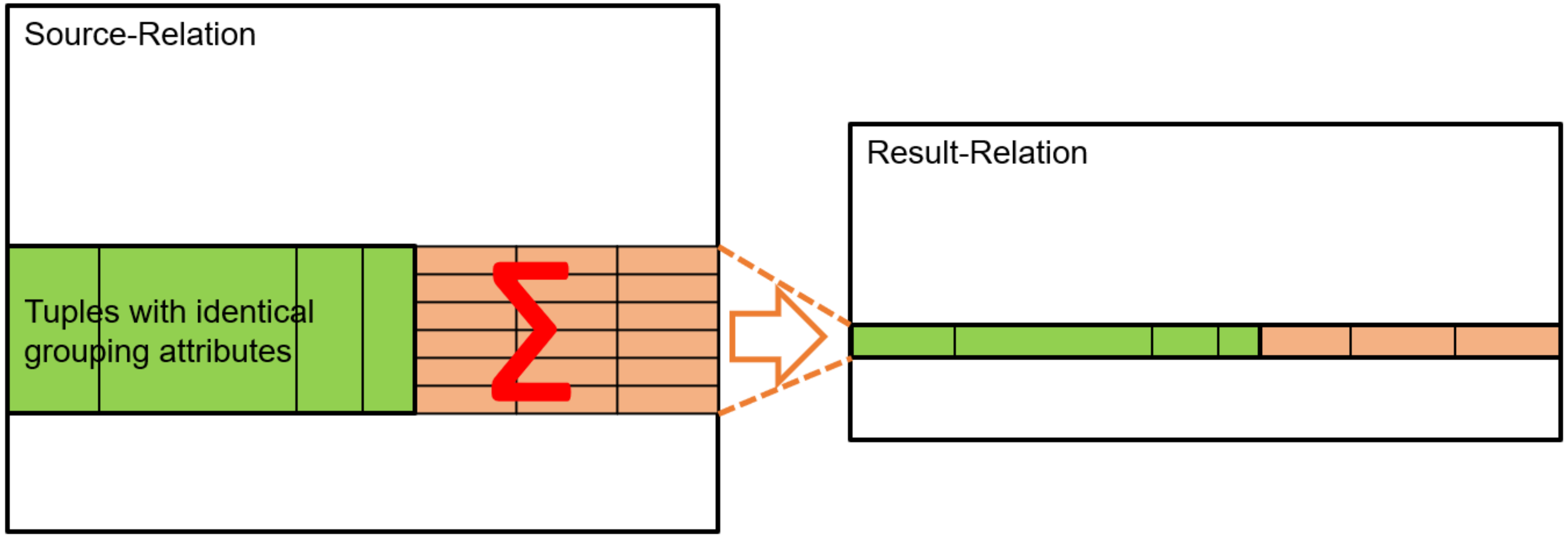
<AxesSubplot:>



Interpolation can also be axis-aware (for time series) or use various methods (e.g. spline, quadratic, piecewise polynomial).

Grouping and aggregation

We look at all tuples having the identical values for all **grouping attributes** and make one new tuple for them.
Each **non-grouping attributes** can be aggregated to one single value (per group).




```
SELECT a,b,c, SUM(…
FROM …

GROUP BY a,b,c
```
SQL

Titanic data set

https://hbiostat.org > **Titanic5** data set

(legend can be found in the titanic.xlsx file)

```
# only passengers (to show CSV import)
# url = 'https://hbiostat.org/data/repo/titanic5.csv'
# titanic = pd.read_csv(url)

# better read from the MS Excel file
url = 'https://hbiostat.org/data/repo/titanic5.xlsx'
titanic = pd.read_excel(url, sheet_name="Titanic5_all")
titanic.head()
```

✓ 3.4s

```
:\Users\Student\AppData\Local\Temp\ipykernel_3312\687780751.py:7: FutureWarning: Inferring datetime64[ns] from data containing strings is deprecated and will be remo
ersion. To retain the old behavior explicitly pass Series(data, dtype=datetime64[ns])
 titanic = pd.read_excel(url, sheet_name="Titanic5_all")
```

	Name_ID	Name	Female	Male	Sex	Age	Class/Dept	Class	Ticket	Joined	...	Title	First	DoB	Year_Birth	Date_Death	DoB_Clean	Age_F_Code
0	531	DEAN, Miss Elizabeth Gladys 'Millvina'	1	0	female	0.17	3rd Class Passenger	3	2315	Southampton	...	Miss	Elizabeth Gladys 'Millvina'	1912-02-02	NaN	2009-05-31 00:00:00	1912-02-02 00:00:00	C
1	498	DANBOM, Master Gilbert Sigvard Emanuel	0	1	male	0.33	3rd Class Passenger	3	347080	Southampton	...	Master	Gilbert Sigvard Emanuel	1911-11-16	NaN	1912-04-15 00:00:00	1911-11-16 00:00:00	C

Variable	Description	
Name_ID	Unique ID for Passenger / Crew member	
Name	Full Name (LAST, Title First)	
Female	Female indicator	
Male	Male indicator	
Sex	Sex (text, male/female)	
Age	Age, numeric (fractional - 10 months = 0.8333)	
Class/Dept	Text field for passenger class or crew department	
Class	1^{st}/2^{nd}/3rd Class Passenger	1/2/3
	Deck Crew	D
	Engineering Crew	E
	Band	B
	Restaurant Staff	R
	Victualling Crew	V
Ticket	Ticket number	
Joined	Port of embarcation	
Occupation	Job / Career	
Boat [Body]	Boat (rescued survivor), Body (identified victim)	
Price	Price of ticket, Pounds	
Job	Second field of career info / company, role, various	
Survived	Survived indicator	

Variable	Description
URL	Encylopedia Titanic URL filename (starts with http://www.encyclopedia-titanica.org)
Last	Last Name
Title	Title / Salutation
First	First Name
DoB	Date of birth if available
Year_Birth	Year or Year/Month of birth if available
Date_Death	Date of death
DoB_Clean	Date of birth, cleaned, derived when only year or month/year of birth available, supplemented with values from "Age" variable if "DoB" or "Year_Birth" not available
Age_F_Code	A code to indicate the method by which Age_F was derived
Age_F	Age "Final" - all ages for which any data exists, see "Age_F_Code" for how it was derived
sibsp	Number of Siblings/Spouses Aboard - obtained from familiar "Titanic3" dataset, merged via "Name_ID" varibale
parch	Number of Parents/Children Aboard - obtained from familiar "Titanic3" dataset, merged via "Name_ID" varibale

Range of values

Simple trick to find out all occuring values in a column: convert the column to a Python **set**:

```
set(titanic["Sex"])
```

```
{'female', 'male'}
```

```
set(titanic["Embarked"])
```

```
{'C', 'Q', 'S', nan}
```

Grouping

The simplest form of grouping is done by specifying the grouping attributes (the "keys"):

```
grouped = t.groupby(by=["Pclass", "Sex"])
grouped
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002028BD33190>
```

This returns a pandas "**GroupBy**" object.

It has the groupings calculated. For each group, it contains a DataFrame with all (!) attributes of the original. Sorting within the groups is preserved.

You can display this with the `groups`-Attribute: it is a Dictionary of the grouping keys and the indexes of that group.

```
grouped.groups
```

```
{(1, 'female'): [1, 3, 11, 31, 52, 61, 88,
307, 309, 310, 311, 318, 319, 325, 329, 3
7, 539, 540, 556, 558, 571, 577, 581, 585
```

That helps to output the groups alone:

```
grouped.groups.keys()
```

```
dict_keys([(1, 'female'), (1, 'male'), (2, 'female'), (2, 'male'), (3, 'female'), (3, 'male')])
```

We can use projection on all grouping and non-grouping attributes. (We use `head()` to display something for each group.)

```
grouped[["Pclass", "Sex", "Age", "Cabin", "Name"]].head(1)
```

	Pclass	Sex	Age	Cabin	Name
0	3	male	22.0	NaN	Braund, Mr. Owen Harris
1	1	female	38.0	C85	Cumings, Mrs. John Bradley (Florence Briggs Th...
2	3	female	26.0	NaN	Heikkinen, Miss. Laina
6	1	male	54.0	E46	McCarthy, Mr. Timothy J
9	2	female	14.0	NaN	Nasser, Mrs. Nicholas (Adele Achem)
17	2	male	NaN	NaN	Williams, Mr. Charles Eugene

```
len(grouped)
```

```
6
```

Grouping

We can iterate through the groups

```python
for (pclass,sex), group in grouped:
    print()
    print(pclass, '-', sex)
    print(group.drop(columns=["Name","SibSp","Parch","Ticket"]))
```

```
1 - female
     PassengerId  Survived  Pclass     Sex   Age      Fare Cabin Embarked
1              2         1       1  female  38.0   71.2833   C85        C
3              4         1       1  female  35.0   53.1000  C123        S
11            12         1       1  female  58.0   26.5500  C103        S
31            32         1       1  female   NaN  146.5208   B78        C
52            53         1       1  female  49.0   76.7292   D33        C
..           ...       ...     ...     ...   ...       ...   ...      ...
856          857         1       1  female  45.0  164.8667   NaN        S
862          863         1       1  female  48.0   25.9292   D17        S
871          872         1       1  female  47.0   52.5542   D35        S
879          880         1       1  female  56.0   83.1583   C50        C
887          888         1       1  female  19.0   30.0000   B42        S

[94 rows x 8 columns]

1 - male
     PassengerId  Survived  Pclass   Sex   Age      Fare        Cabin Embarked
6              7         0       1  male  54.0   51.8625          E46        S
23            24         1       1  male  28.0   35.5000           A6        S
27            28         0       1  male  19.0  263.0000  C23 C25 C27        S
30            31         0       1  male  40.0   27.7208          NaN        C
34            35         0       1  male  28.0   82.1708          NaN        C
..           ...       ...     ...   ...   ...       ...          ...      ...
```

```python
print(type(group))
```

```
<class 'pandas.core.frame.DataFrame'>
```

We can select a single group

```python
grouped.get_group((3, "male"))
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S
12	13	0	3	Saundercock, Mr. William Henry	male	20.0	0	0	A/5. 2151	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...	...

Aggregation

The size of each group (including Nulls)

```
grouped.size()
```

```
Pclass  Sex
1       female     94
        male      122
2       female     76
        male      108
3       female    144
        male      347
```

```
SELECT "Pclass" ", "Sex", COUNT(*)
FROM "titanic"
GROUP BY "Pclass", "Sex";
```
SQL

The number of non-Null values for "Age" of each group

```
grouped["Age"].count()
```

```
Pclass  Sex
1       female     85
        male      101
2       female     74
        male       99
3       female    102
        male      253
```

```
SELECT "Pclass" ", "Sex", COUNT("Age")
FROM "titanic"
GROUP BY "Pclass", "Sex";
```
SQL

Some standard aggregation methods:

`size`	Number of rows in group
`count`	Number of non-Null values
`sum`	Sum of non-Null values
`mean`	Average of non-Null values
`median`	Median of non-Null values
`min, max`	Minimum, maximum of non-Null values
`nunique`	Number of distinct values

Aggregation

To have more control and do multiple aggregations at the same time, the following syntax helps:

```python
grouped.agg(
    nof_persons = ("Survived", "size"),
    nof_survived = ("Survived", "sum"),
    avg_fare = ("Fare", "mean"),
    min_fare = ("Fare","min")
).assign(survival_rate = lambda grp: grp["nof_survived"] / grp["nof_persons"])
```

Pclass	Sex	nof_persons	nof_survived	avg_fare	min_fare	survival_rate
1	female	94	91	106.125798	25.9292	0.968085
	male	122	45	67.226127	0.0000	0.368852
2	female	76	70	21.970121	10.5000	0.921053
	male	108	17	19.741782	0.0000	0.157407
3	female	144	72	16.118810	6.7500	0.500000
	male	347	47	12.661633	0.0000	0.135447

```sql
SELECT "Pclass", "Sex",
    COUNT(*) AS "nof_persons,
    SUM("Survived") AS "nof_survived",
    AVG("Fare") AS "avg_fare",
    MIN("Fare") AS "min_fare",
    SUM("Survived")/COUNT(*) AS "survival_rate"
FROM "titanic"
GROUP BY "Pclass", "Sex";
```

SQL

Special cases

No aggregation – return only the groups

Several rather complicated solutions…

```
grouped.first().reset_index()[["Pclass", "Sex"]]
```

	Pclass	Sex
0	1	female
1	1	male
2	2	female
3	2	male
4	3	female
5	3	male

```
SELECT "Pclass", "Sex"
FROM "titanic"
GROUP BY "Pclass", "Sex";
```
SQL

```
titanic.drop_duplicates(subset=["Pclass", "Sex"])[["Pclass", "Sex"]]
```

```
grouped[["Pclass", "Sex"]].head(1)
```

```
grouped.head(1)[["Pclass", "Sex"]]
```

```
pd.DataFrame(grouped.groups.keys(), columns=["Pclass", "Sex"])
```

`grouped.groups.keys()` is my favorite.

No grouping – do aggregation over all rows

Use the aggregation functions on a column

```
(
    titanic["Age"].mean().round(1),
    titanic["Survived"].sum()
)
```

```
(29.7, 342)
```

```
SELECT AVG("Age"),
       SUM("Survived")
FROM "titanic";
```
SQL

Warning: check whether or not the desired function is defined on your pandas-object (SeriesGroupBy, DataFrame,..).

Grouping and Nulls

Building the groups

pd.NA usually is not a group unless we wish it to be so.

```
titanic.groupby("Age", dropna=False).size()
```

```
Age
0.42       1
0.67       1
0.75       2
0.83       2
0.92       1
        ...
70.50      1
71.00      2
74.00      1
80.00      1
NaN      177
```

Caution: The group is named np.NaN (I would expect pd.NA instead).

Aggregation within a group

Repetition: Aggregation methods ignore Null values.

Some aggregation methods do have an option to change this behaviour to `skipna=False`

Aggregation methods

In SQL, we specify a non-grouping attribute plus an aggregation function, e.g. `SUM(age)`. The result is one single value (per group).

To give an overview of aggregation methods in pandas, we differentiate according to the sizes of the input / the result and how the calculation is done.

a) We start with most basic method, which returns one value per group, but does not really look into any values of the group.

`size`	Number of rows in group

b) for each given column(s), based on the values of that column, calculate several figures per group, resp.

`describe`	(count, mean, std, min, 25%, 50%, 75%, max) statistics for each group

Aggregation methods

c) for each given column(s), based on the values of that column, calculate one value per group, resp.

`count`	Number of non-Null values
`sum`	Sum of non-Null values
`prod`	Product of non-Null values
`min, max`	Minimum, maximum of non-Null values
`mean`	Average of non-Null values
`median`	Median of non-Null values
`std, var, skew, kurt, mad`	Standard deviation, variance (n-1 in denominator), skew, kurtosis, mean absolute deviation
`quantile(0.9)`	90 % quantile

Sometimes it is possible to apply a method to all columns of the groupby-object, sometimes we have to project to some columns first.

```
grouped.min()
```

```
ValueError: Wrong number of items passed 2, placement implies 4
```

```
grouped["Age"].min()
```

```
Pclass  Sex
1       female    2.00
        male      0.92
2       female    2.00
        male      0.67
3       female    0.75
        male      0.42
Name: Age, dtype: float64
```

Sometimes, columns with unsuitable dtype are silently dropped.

```
grouped.count()
```

Pclass	Sex	PassengerId	Survived	Name	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	female	94	94	94	85	94	94	94	94	81	92
	male	122	122	122	101	122	122	122	122	95	122
2	female	76	76	76	74	76	76	76	76	10	76
	male	108	108	108	99	108	108	108	108	6	108
3	female	144	144	144	102	144	144	144	144	6	144
	male	347	347	347	253	347	347	347	347	6	347

```
grouped.sum()
```

Pclass	Sex	PassengerId	Survived	Age	SibSp	Parch	Fare
1	female	44106	91	2942.00	52	43	9975.8250
	male	55599	45	4169.42	38	34	8201.5875
2	female	33676	70	2125.50	37	46	1669.7292

Aggregation methods

d) for each given column(s), search within the values of the column and return one value

`idxmin`, `idxmax`	Return the index value (within the original non-grouped DataFrame) of the first row having the minimal value for the group.
`nlargest`	Largest value of one column within group
`argmin`, `argmax`	Caution: This method is not available for groups, it only works on entire (non-grouped) DataFrame. Return the position (starting with 0) of the first row having the minimal value.

The overall youngest passenger is

```
titanic["Age"].argmin()
```

```
803
```

```
titanic.iloc[803][["PassengerId", "Name", "Age"]]
```

```
PassengerId                              804
Name           Thomas, Master. Assad Alexander
Age                                     0.42
Name: 803, dtype: object
```

Example:

The index of the youngest passenger within each group is

```
idx = grouped["Age"].idxmin()
print(idx)
```

```
Pclass  Sex
1       female    297
        male      305
2       female    530
        male      755
3       female    469
        male      803
Name: Age, dtype: int64
```

and print the related passenger for each group

```
titanic.loc[idx][["Sex", "Pclass", "PassengerId", "Name", "Age"]]
```

	Sex	Pclass	PassengerId	Name	Age
297	female	1	298	Allison, Miss. Helen Loraine	2.00
305	male	1	306	Allison, Master. Hudson Trevor	0.92
530	female	2	531	Quick, Miss. Phyllis May	2.00
755	male	2	756	Hamalainen, Master. Viljo	0.67
469	female	3	470	Baclini, Miss. Helene Barbara	0.75
803	male	3	804	Thomas, Master. Assad Alexander	0.42

Aggregation methods

Test to demonstrate that `idxmin()` is really returning the index, not the position:

```
t = titanic.set_index(["Cabin", "Sex", "PassengerId"])
print(f'Index is unique: {t.index.is_unique}')
idx = t.groupby(["Sex", "Pclass"])["Age"].idxmin()
print(idx)

t.loc[idx].reset_index()[["PassengerId", "Name", "Age"]]
```

```
Index is unique: True
Sex     Pclass
female  1         (C22 C26, female, 298)
        2             (nan, female, 531)
        3             (nan, female, 470)
male    1           (C22 C26, male, 306)
        2               (nan, male, 756)
        3               (nan, male, 804)
Name: Age, dtype: object
```

	PassengerId	Name	Age
0	298	Allison, Miss. Helen Loraine	2.00
1	531	Quick, Miss. Phyllis May	2.00
2	470	Baclini, Miss. Helene Barbara	0.75
3	306	Allison, Master. Hudson Trevor	0.92
4	756	Hamalainen, Master. Viljo	0.67
5	804	Thomas, Master. Assad Alexander	0.42

Aggregation methods

e) for the total of all projected columns, select / compute some information within the group.

`first, last, nth`	First, last, nth value within group according to index sorting
`head`	first n rows within group according to index sorting `head(1)` is not the same as `first()`: the index of the result is different (see later)
`any`	Return `True` when any of the values within the group (for all selected rows and columns of the group) is truthful ("True" Boolean or non-zero number or non-empty text).
`all`	Return `True` when all values within the group are truthful.

```
grouped[["Name", "Sex"]].first()
```

Pclass	Sex	Name	Sex
1	female	Cumings, Mrs. John Bradley (Florence Briggs Th...	female
	male	McCarthy, Mr. Timothy J	male
2	female	Nasser, Mrs. Nicholas (Adele Achem)	female
	male	Williams, Mr. Charles Eugene	male
3	female	Heikkinen, Miss. Laina	female
	male	Braund, Mr. Owen Harris	male

```
grouped[["Name", "Sex"]].head(1)
```

	Name	Sex
0	Braund, Mr. Owen Harris	male
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female
2	Heikkinen, Miss. Laina	female
6	McCarthy, Mr. Timothy J	male
9	Nasser, Mrs. Nicholas (Adele Achem)	female
17	Williams, Mr. Charles Eugene	male

~~Aggregation~~ transformation methods

f) return the original DataFrame (not grouped any more), but after some computation within the groups took place

`cumsum`	Compute the cumulated sum within each group
`cumprod`	Same with cumulated product
`cummin`, `cummax`	Cumulated minimum / maximum, i.e. the minimum of the first i rows.
`diff`	Difference to value in previous row (result for first row is np.nan)
`pct_change`	Percentage change to value in previous row (result for first row is pd.NA)

```
data = {"x" : [10,1,3,21]}
df = pd.DataFrame(data)
df["cumsum(x)"] = df["x"].cumsum()
df["cummin(x)"] = df["x"].cummin()
df["diff(x)"] = df["x"].diff()
df["pct_change(x)"] = df["x"].pct_change()*100
df
```

	x	cumsum(x)	cummin(x)	diff(x)	pct_change(x)
0	10	10	10	NaN	NaN
1	1	11	1	-9.0	-90.0
2	3	14	1	2.0	200.0
3	21	35	1	18.0	600.0

The effect of cumulated sums with the groups is not obvious in the ordering of the original DataFrame.

```
grouped.cumsum()
```

	PassengerId	Survived	Age	SibSp	Parch	Fare
0	1	0	22.00	1	0	7.2500
1	2	1	38.00	1	0	71.2833
2	3	1	26.00	0	0	7.9250
3	6	2	73.00	2	0	124.3833
4	6	0	57.00	1	0	15.3000
...	...	...	...	...	...	...

After reordering according to the grouping, the output makes more sense

```
titanic["cumsum(Age)"] = grouped["Age"].cumsum()
titanic.set_index(["Pclass", "Sex"]).sort_index()[["Age", "cumsum(Age)"]].head(3)
```

		Age	cumsum(Age)
Pclass	**Sex**		
1	**female**	38.0	38.0
	female	35.0	73.0
	female	58.0	131.0

Grouping and the index

Usually the aggregation returns a DataFrame having an index according to the grouping keys.

```
grouped["Fare"].sum().index
```

```
MultiIndex([(1, 'female'),
            (1,   'male'),
            (2, 'female'),
            (2,   'male'),
            (3, 'female'),
            (3,   'male')],
           names=['Pclass', 'Sex'])
```

But for some methods the index of the original non-grouped DataFrame is returned, namely for head() and the methods in f)

```
grouped["Fare"].cumsum().index
```

```
RangeIndex(start=0, stop=891, step=1)
```

To get rid of the index columns, you would call `reset_index()` afterwards…

```
grouped["Fare"].sum().reset_index()
```

	Pclass	Sex	Fare
0	1	female	9975.8250
1	1	male	8201.5875
2	2	female	1669.7292
3	2	male	2132.1125
4	3	female	2321.1086
5	3	male	4393.5865

or use `as_index=False` already when creating the groupings.

```
titanic.groupby(["Pclass", "Sex"], as_index=False)["Fare"].sum()
```

	Pclass	Sex	Fare
0	1	female	9975.8250
1	1	male	8201.5875
2	2	female	1669.7292
3	2	male	2132.1125
4	3	female	2321.1086
5	3	male	4393.5865

Free aggregation / transformation methods

g) Apply any function f to the groups.
f should return

i) a <u>scalar</u> (thus it is an aggregation reducing each group to one value), or,

ii) a <u>DataFrame</u> or Series; the individual group results will then be assembled using `concat()`

Example: Use the Series-function argmin() on groups to find out which row attains the minimum fare.

```
grouped["Fare"].min()
```

```
Pclass  Sex
1       female    25.9292
        male       0.0000
2       female    10.5000
        male       0.0000
3       female     6.7500
        male       0.0000
```

```
grouped["Fare"].apply(pd.Series.argmin)
```

```
Pclass  Sex
1       female     81
        male       32
2       female      5
        male       37
3       female    113
        male       74
```

Check that 5 is indeed the row with the minimum fare:

```
grouped.get_group((2, "female")).iloc[5]["Fare"]
```

```
10.5
```

Example: Replace missing values with the average within each group.

```
df
```

	x	y
A	1	<NA>
A	<NA>	<NA>
B	42	<NA>
A	5	77
B	42	<NA>

```
fill_mean = lambda g: g.fillna(g.mean())
df.groupby(df.index).apply(fill_mean)
```

		x	y
A	**A**	1.0	77.0
	A	3.0	77.0
	A	5.0	77.0
B	**B**	42.0	NaN
	B	42.0	NaN

Caution: `apply()` works differently on groups and a df

`df.groupby(…).apply(f)`

Here, f is applied to all groups, i.e. for each group we do one call to f(g) where g is the df of the group.

f: df $\longmapsto$ x (DataFrame is mapped to a number)

```
def weighed_fare(df):
    total = df["Fare"].sum()
    nof_kids = df["Parch"].count() # questionable
    return total / nof_kids
```

```
weighed_fare(titanic)
```

```
32.204207968574636
```

```
grouped.apply(weighed_fare)
```

```
Pclass  Sex
1       female    106.125798
        male       67.226127
2       female     21.970121
        male       19.741782
3       female     16.118810
        male       12.661633
```

`df.apply(f, axis='columns')`

When we work on a df without grouping, then f is called for each <u>row</u> of the df.

f: row $\longmapsto$ x (row of DataFrame is mapped to a number)

```
titanic.apply(
    lambda row: row["Fare"] / row["Age"]
    , axis='columns'
)
```

```
0       0.329545
1       1.875876
2       0.304808
3       1.517143
4       0.230000
          ...
```

Warning: apply() is inherently <u>slow</u>, because it executes a function on each and every row. Better is to use a vectorized operation (based on aligning with the help of an index) if possible.

h) Transform all values group wise, returning a DataFrame with the <u>same shape</u>. It is comparable to f) before, but transformed using some arbitrary function.

```
titanic["survived_per_class"] = (
    titanic.groupby("Pclass")["Survived"].transform('count')
)
titanic[["Pclass", "Name", "survived_per_class"]].head()
```

	Pclass	Name	survived_per_class
0	3	Braund, Mr. Owen Harris	491
1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	216
2	3	Heikkinen, Miss. Laina	491
3	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	216
4	3	Allen, Mr. William Henry	491

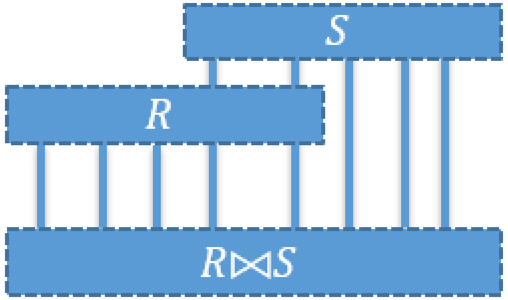
You might also use an arbitrary function instead of standard aggregations like `count`.

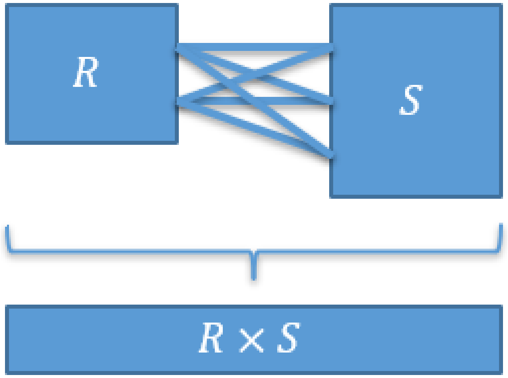
There are more methods, we haven't mentioned yet.
Just a (growing) list for inspiration and my relief:

`rank`	Provide a rank number based on the sorted values
`unique`	Returns the DataFrame with duplicates removed
`filter`	Return rows belonging to groups which fulfil the filter condition. This is not like SQL-HAVING which returns the filtered groups. Rather, it is a selection in the original DataFrame – see f) before.

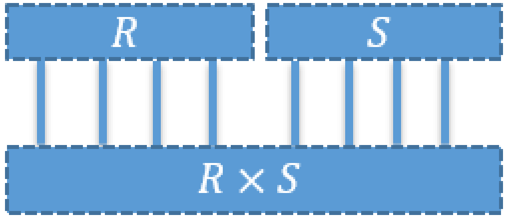
Joins

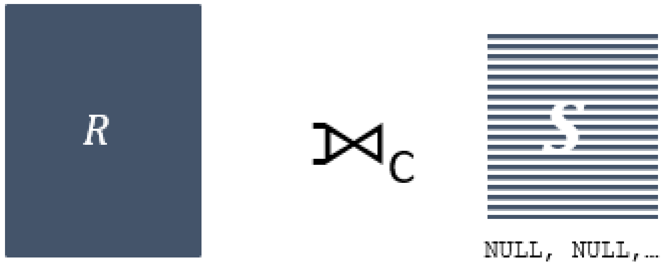
Combining DataFrames





Relational algebra	SQL
$R \bowtie S$	SELECT … FROM R **NATURAL JOIN S** WHERE …
$R \bowtie_{\text{<condition>}} S$	SELECT … FROM R **JOIN S ON** <condition> WHERE …
R ⟕$_{\text{<condition>}}$ S	SELECT … FROM R **LEFT OUTER JOIN S ON** <condition> WHERE …





Equijoin without using index

First, we need a DataFrame to join to:

```
data = {"town_abbrev": ['C', 'Q', 'S'],
        "town": ['Cherbourg', 'Queenstown', 'Southampton'],
        "country": ['France', 'Ireland', 'England']}
towns = pd.DataFrame(data).convert_dtypes()
towns
```

	town_abbrev	town	country
0	C	Cherbourg	France
1	Q	Queenstown	Ireland
2	S	Southampton	England

Now, do the join using `merge()`

```
titanic2 = titanic.merge(towns, how="inner", left_on="Embarked", right_on="town_abbrev")
titanic2.head()
```

Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	town_abbrev	town	country
Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	S	Southampton	England
iss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	S	Southampton	England
ques el)	female	35.0	1	0	113803	53.1000	C123	S	S	Southampton	England
y	male	35.0	0	0	373450	8.0500	NaN	S	S	Southampton	England
/ J	male	54.0	0	0	17463	51.8625	E46	S	S	Southampton	England

Observations:

- Index of "titanic" vanished. We have a new RangeIndex.
- All joined columns are preserved in the result ("Embarked" and "town_abbrev").

```
SELECT *
FROM "titanic"
JOIN "towns" ON "titanic"."Embarked" = "towns"."town_abbrev";
```

SQL

Left outer join

The inner join "lost" some passengers where we don't know the town of embarkment:

```
dict(titanic=titanic.shape, joined=titanic2.shape)
```

```
{'titanic': (891, 12), 'joined': (889, 15)}
```

```
titanic.groupby("Embarked", dropna=False).size()
```

```
Embarked
C      168
Q       77
S      644
NaN      2
```

Remedy is a left outer join

```
titanic3 = titanic.merge(towns, how="left", left_on="Embarked", right_on="town_abbrev")
titanic3.shape
```

```
(891, 15)
```

```
titanic3.loc[titanic3["Embarked"].isna()]
```

Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	town_abbrev	town	country
Miss. Amelie	female	38.0	0	0	113572	80.0	B28	NaN	<NA>	<NA>	<NA>
rtha elyn)	female	62.0	0	0	113572	80.0	B28	NaN	<NA>	<NA>	<NA>

Equijoin using index

Define the index on the right (lookup) table:

```
towns.set_index("town_abbrev", inplace=True)
towns
```

	town	country
town_abbrev		
C	Cherbourg	France
Q	Queenstown	Ireland
S	Southampton	England

In the (left) "titanic" table, an index on "PassengerId" would make sense, but is not helpful in this situation.

```
titanic.merge(towns, left_on="Embarked", right_index=True)
```

Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	town	country
ınd, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	Southampton	England
ɘn, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	Southampton	England
ıeath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S	Southampton	England
villiam Henry	male	35.0	0	0	373450	8.0500	NaN	S	Southampton	England

Observations:

- It is a little bit confusing, what columns survive and what the final index is in this situation (a mixture of joining by attribute(s) in one df and the index in the other df).
- Usually, we would do `left_on`/`right_on`, or, use the index in <u>both</u> df.

Remarks on joining

Observations

- Only Equijoin is possible using `merge()` !
- Specify the columns for the join condition:
  - Use `left_on=` or `left_index=True` for the left table
  - Use `right_on=` or `right_index=True` for the right table
  - Use `on=` when the column names match in both df's
  - When you omit to specify the join columns, a natural join is done (join columns with identical names in both tables)
- When using an index in both df to join, the index column(s) are preserved in the result.
  In contrast, if you do not use the index as a join condition (`left_on=` / `right_on=`), then the index is ignored and lost! If you need the index later, you need to preserve it before: `df["index"] = df.index`
  Don't make assumptions about row order in the result.
- To avoid ambiguity for identical column names in both tables, a suffix "`_x`" / "`_y`" is appended (can be defined).

Hints

- dtypes of joined columns should match, otherwise you need to adapt them before joining.
- Check out additional options of `merge()`
- `validate="1:1"` (or 1:m, m:1) checks the cardinalities of the join
- Forget the `join()` method. It is just a wrapper around `merge()` to save some typing if you want to join along an index.
- Forget also `concat(…, axis='columns')`. This also glues two tables together horizontally based on matching column names (like a natural join in SQL). However, `concat()` comes handy, when you "join" multiple Series into a df.

RDBMS versus pandas

RDBMS

pandas

DML operations

To complete the introduction along the lines of SQL, we need to look at some more commands. Let's start with INSERT, UPDATE, DELETE.

How to do an **INSERT**?

We could create a new DataFrame with the desired information and append it. This can be done using `concat()`.

UPDATE

We already did updates by computing new columns (extended projection).

How to mimic the WHERE-condition?

We can select the rows first using `loc[]` and then `apply()` the intended function.

DELETE

Do a selection of the rows you don't want to delete.

Example:

```
o = o.loc[o.country.isin(['DE','DK'])]
```

Set operations

UNION / UNION ALL

Use `concat()`

Warning: `append()` is just a special version of `concat`, but not really as efficient, since it creates a new DataFrame each time.

MINUS

There is no fitting command.

If we want to do a minus just with respect to one single column, we could filter for values not occurring in the other set, like so:

```
df_a.loc[df_a["col"].isin(df_b["col"]) == False]
```

INTERSECT

There are no direct equivalents as to my knowledge.

But there is a method doing the `intersection()` of a (multi) index, which could be used to solve one task.

A solution might also be to do a suitable inner join instead.

Subqueries can be used at several places within an SQL.

Nested subquery in FROM clause

Example:

```
WITH job_durations
AS (
   SELECT last_name AS name
      , end_date - start_date AS duration
   FROM hr.job_history j
   JOIN hr.employees e ON j.employee_id = e.employee_id
)
SELECT name, COUNT(*), AVG(duration)
FROM job_durations
GROUP BY name;
```

SQL

Solution: Compute the FROM-clause first, and then continue with the outer SELECT.

Observation: In contrast to SQL, we have to materialize the intermediate results.

Subquery in the SELECT-list

Subquery in WHERE clause

We have to distinguish certain cases: is the shape of the subquery 1x1, 1xn, 2xn,…?

IN / EXISTS / ALL / ANY comparisons.

Correlated subqueries

All this is getting more complex and there are no simple recipes. Solutions include translating it into a join expression first (which usually will do the RDBMS for you).

The ultimate comparison

RDBMS

- SQL has few but powerful concepts.
- Data structure is the relation.
- Work is row-oriented (first filter based on WHERE condition and then do modifications on that data).
- Indexes are used for performance tuning (and unique constraints).
- Supports transactional use and hence comes with an overhead in terms of operation and resources.

pandas

- pandas has a vast number of very specific commands → more to learn, tricky to use (and read), solutions can be elegant and quick.
- Data structures are DataFrame, Series, GroupBy and other object types.
- Work is more column-oriented (manipulate entire columns = Series at once with vector operations).
- Index is used to align data to facilitate further operations. There is only one (row) index.
- pandas is open source – for commercial support use e.g. Anaconda distribution
- Optimized for (one-shot) data analysis of static files. In this application domain it has advantages: easy to set up, faster to develop, nice documentation with Jupyter notebooks.

Memory usage

Working with large datasets or creating too many copies of the data may cause a memory error.

```
import pandas as pd
```

```
d = pd.read_csv("OrgxProj Geocodes.csv")
```

```
x = pd.concat([d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d,d])
```

```
y = pd.concat([x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x])
```

```
z = pd.concat([y,y,y,y,y,y,y,y,y,y,y,y,y])
```

```
z.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 318113928 entries, 0 to 23626
Data columns (total 16 columns):
 #   Column           Dtype
---  ------           -----
 0   year             float64
 1   project id       int64
 2   project acronym  object
 3   project cost     float64
 4   role             object
 5   org id           int64
 6   org name         object
 7   country          object
 8   post code        int64
 9   city             object
 10  street           object
 11  org url          object
 12  lat              float64
 13  lng              float64
 14  places_found     int64
 15  place_id         object
dtypes: float64(4), int64(4), object(8)
memory usage: 40.3+ GB
```

After DataFrame `z`, the Python process had already 9 GiB.

Name	Status	35% CPU	78% Arbeitssp...
Python		0%	8.739,0 MB
Python		0%	15,9 MB

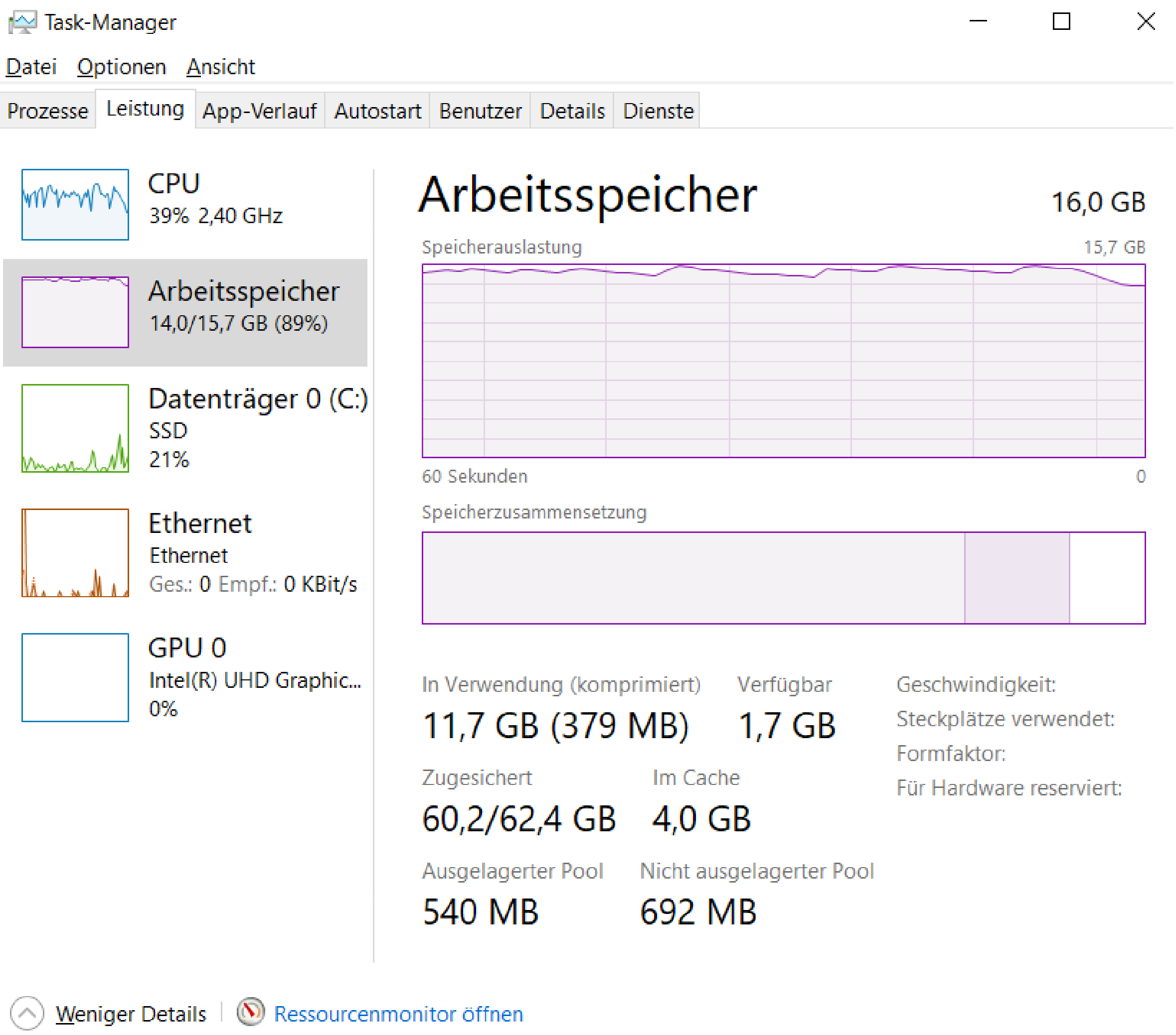
The next step crashes:

```
t = pd.concat([z,z,z,z,z,z,z,z,z,z,z,z,z,z,z,z,z])
```

```
MemoryError: Unable to allocate 9.48 GiB for an array with shape (4, 318113928) a
```

Memory usage

My PC with 16 GB RAM tried to swap data to hard disk, but for processing it still needs to go into the RAM. RAM is already some time on the top edge before the crash.



To clean up the memory you can delete objects.

```
del(y)
del(z)
del(t)
```

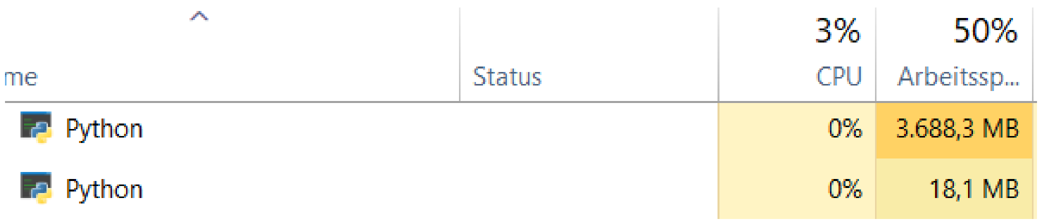
After the automatic run of the Python garbage collector, the space will be freed.

You can force the garbage collector run using

```
import gc
```

```
gc.collect()
```

Still this does not mean that all memory is returned to the system. Although the memory is free now, Python keeps a portion of it – just in case.



Scalability and performance

As we have seen, scalability is an issue in pandas. For larger data sets, performance is an issue, too.

As a symptom for this situation, there are various solutions to it.

Tricks within **pandas**:

- Load only needed data by filtering columns in `read_csv(…, usecols=… )`
- Use efficient data types
- You can read and process data files in chunks using `read_parquet()`
- Enhancing performance by programming extension in C or other advanced approaches, see: https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html

Outside pandas, currently there are:

Project **ibis** to do SQL-like operations on various backends including pandas, sqlalchemy (hence RDBMS) and Spark. But this would mean to work with ibis instead of pandas.

Use Apache **Spark** instead of pandas. It is a cluster computing platform also providing "DataFrames" and SQL.

Use Apache Spark as a backend for pandas. Apache **arrow** (pyarrow) is the bridge between Python and the Spark JVM process. https://towardsdatascience.com/spark-vs-pandas-part-4-recommendations-35fc554573d5

Most promisingly: **dask** allows to scale pandas DataFrames. The dask DataFrame and commands very much look like pandas, but they also live on disk or a remote computer. New: **polars** is comparable to pandas, but support parallel execution on CPU cores.

SAP HANA

[1]:
```
import hana_ml.dataframe as dataframe
conn = dataframe.ConnectionContext(userkey='MYHANA')
df_remote = conn.table('USEDCARS')
```

Lassen Sie sich einige wenige Zeilen der Tabelle anzeigen.

[2]:
```
df_remote.head(5).collect()
```

[2]:

	CAR_ID	BRAND	MODEL	VEHICLETYPE	YEAROFREGISTRATION	HP	FUELTYPE	GEARBOX
0	717	volkswagen	golf	limousine	2004	75	benzin	manuell
1	860	volkswagen	golf	limousine	1999	75	benzin	manuell
2	1557	volkswagen	golf	limousine	1999	75	benzin	manuell
3	1891	volkswagen	golf	limousine	1996	75	benzin	manuell
4	2326	volkswagen	golf	limousine	2000	75	benzin	manuell

Alternatively, process data within a data base, e.g. SAP HANA.

Library `hana_ml` has pandas-ish commands. With `collect()` retrieve data into a local pandas-df.

Performance and scalability solved.