Python & pandas

A one day course

Uwe Ziegenhagen

`github.com/UweZiegenhagen/OneDayPythonPandasCourse`

Cologne, 15. Juli 2022

Introduction

Why Python/pandas?

- You *have* a CSV-file with semicolon as column separator and comma as decimal separator
- You *need* a CSV-file with comma as column separator and dot as decimal separator

```
import pandas as pd

df = pd.read_csv('myfile.csv', sep=';', decimal = ',')
df = pd.to_csv('myfile_new.csv', sep=',', decimal = '.')
```

Limits of this Course

- Basis for this course is a semester-long course I held at the FOM („Fachhochschule für Ökonomie und Management“) in Cologne
- It is not a *full* Python & pandas course, we would need a whole week or more for this.
- Goal: Teach you enough Python to a) read and b) understand Python-Code and c) write smaller programs relevant for your job
- We will skip many interesting things (that you do not necessarily need for your job)

Introduction

Python
- Datatypes
- Functions
- Flow control

pandas

Links

Python

- Invented by Guido van Rossum at the „Centrum Wiskunde & Informatica“ in Amsterdam as successor for the teaching language ABC
- Current version is 3.11
- For a long time, Python 3.x and Python 2.7 existed together
- Python 2.7 support expired in 2019:
- How to spot 2.7 code: → `print 'hello'` instead of `print('hello')`

Python versus Java & C

Python code is often much slower than C or Java but...

- the implementation time for Python is way faster
- speeds only matters sometimes, not always
- many computing-intensive Python modules use C/C++ modules „under the hood“
- bad C-Code is slower than good Python-code

Python

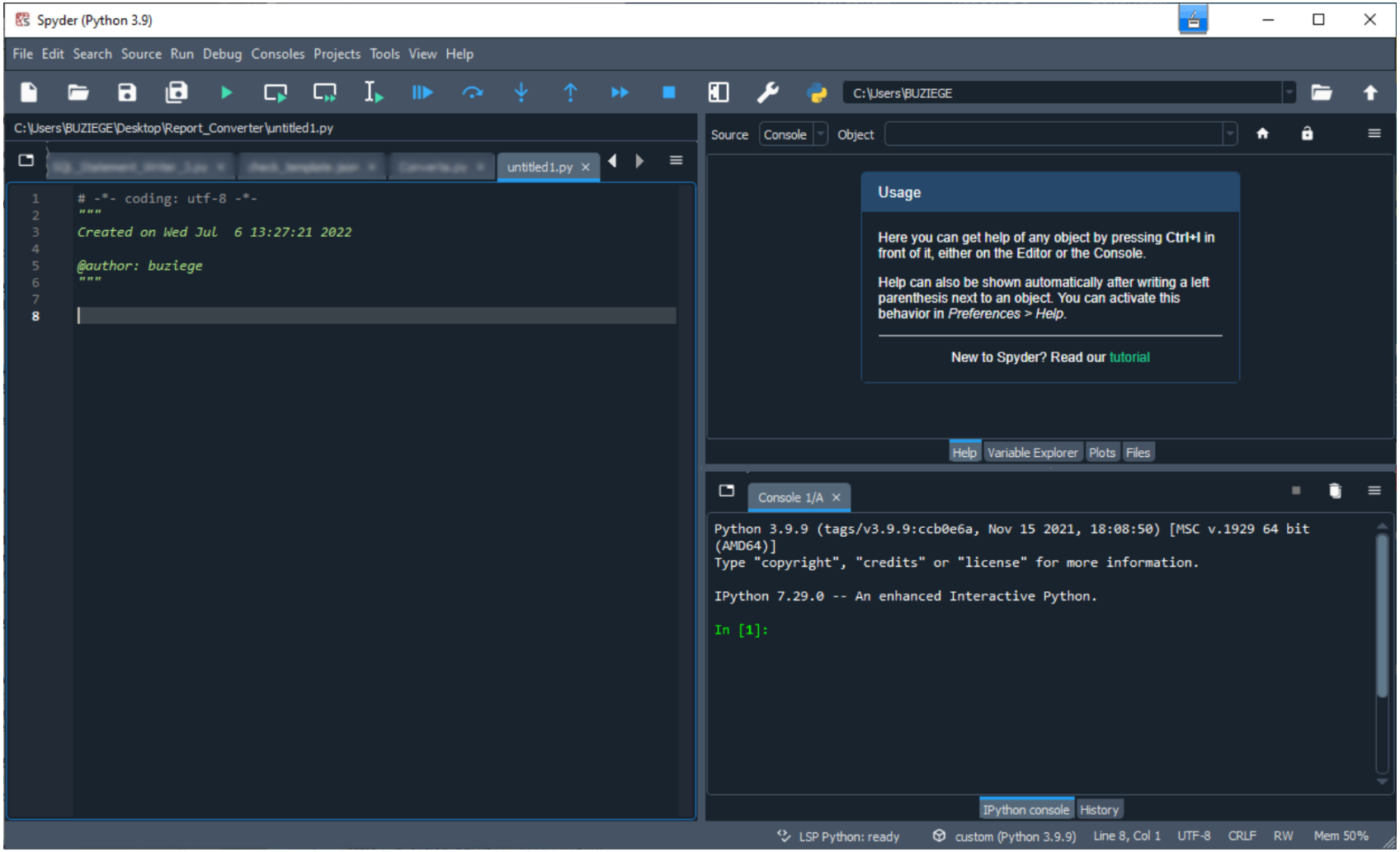
Datatypes

Functions

Flow control

Spyder

- We will use the Spyder5 IDE with Python 3.9.9, make sure it is installed



Python as a Calculator

- Spyder5 runs an IPython kernel, this runs our programs
- We can also use it as a calculator

```
In [1]: 4*5.4
Out[1]: 21.6

In [2]: 4/12
Out[2]: 0.3333333333333333

In [3]: _*3
Out[3]: 1.0

'hello'
Out[4]: 'hello'
```

Python as a Calculator

```

In [1]: 4%2
Out[1]: 0

In [2]: 5%2 # Modulo
Out[2]: 1

In [3]: 3**3
Out[3]: 27

In [4]: 5//2
Out[4]: 2
```

Priority of Operators

- Round brackets have highest priority
- followed by Power
- followed by multiplication and division
- followed by addition and substraction

TODO4U:

$\Rightarrow$ Solve exercise sheet 1!

Basic Input & Output

```
1 print('Hello World')
2
3 yourName = input('Tell me your name: ')
4
5 print('Hello ' + yourName + ', welcome to this class')
6
7 print('Hello ', yourName,', welcome to this class', sep='')
8
9 print(f'Hello {yourName}, welcome to this class')
```

- `input()` only reads strings
- If you need a number, you need to convert it
- there are better ways than `print()` for logging, but it works...
- f-Strings (last row) are recommended for mixed output!

Rules for Variables

- must start with a letter or _
- Case-sensitivity: 'A' is not 'a'
- Recommendation: small letters
- Let them speak for themselves: 'diameter' is good, 'd' is bad

Reserved Keywords

The following keywords are reserviert and must not be used for variables' names.

and	as	assert	break	class
continue	def	del	elif	else
except	False	finally	for	from
global	if	import	in	is
lambda	None	nonlocal	not	or
pass	raise	return	True	try
while	with	yield		

Datentypen

- Integer (integer numbers)
- Float (Floating point number)
- Strings
- Booleans
- Complex numbers

Integer

- unlimited length
- must not start with 0 if they shall represent decimal numbers
- Leading 0 by representation in hex-, binary- or Octal system:

  `0b/0B` binary
  `0x/0X` hex
  `0o/0O` octal

- Functions `hex()`, `bin()`, `oct()` for conversions into string, are internally represented as decimal numbers

Float

- floating point numbers
- `3.1415927`
- `3.1e8`
- Hint: Not every floating point number can be represented *exactly* („Floating-Point Arithmetic“)
- `docs.python.org/3/tutorial/floatingpoint.html`

Strings

- Single or double quotes
- For multiline strings:
  - Triple double or single quotes
  - Alternatively backslash at the end of the line
- Python has numerous functions for strings, more on this later
- Comments start with a hash #

```
a = "I am a string"

b = 'me too'

c = """I am
as string as well """

# I am a comment
```

Booleans

- Named after George Boole
- 1854: „An investigation into the Laws of Thought“
- Essence of modern computing
- Boolean operators or ($\cup$), and ($\cap$), `not`

```
a = True
b = False

a == b #False
a or b # True
a and b # False
a and not b # True
not a and b # False
```

Type Conversions

- Mixing strings and floats/integers requires explicit type conversion using the `str()` function
- Hint: pandas provides `.astype(<Datatype>)`

```
>>> a + str(b)
'abc123'
>>> a+str(c)
'abc3.141'
>>> a*str(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'

>>> 'a' * 3
'aaa'

>>>3 * 'a'
'aaa'
```

Functions

- Functions: named sequence of commands
- Purpose: encapsule code for multiple calls
- Can take arguments as input, can have return values
- Define a new function as
  - Determine name
  - Determine arguments
  - Define function code

Built-in Functions

abs	delattr	hash	memoryview	set
all	dict	help	min	setattr
any	dir	hex	next	slice
ascii	divmod	id	object	sorted
bin	enumerate	input	oct	staticmethod
bool	eval	int	open	str
breakpoint	exec	isinstance	ord	sum
bytearray	filter	issubclass	pow	super
bytes	float	iter	print	tuple
callable	format	len	property	type
chr	frozenset	list	range	vars
classmethod	getattr	locals	repr	zip
compile	globals	map	reversed	__import__
complex	hasattr	max	round	

Functions in C and Python

```c
    #include<stdio.h>
    long add(long a, long b){
        long result;
        result = a + b;
        return result;}

    int main(){
       int a, b, c;
       printf("Enter two numbers\n");
       scanf("%d%d", &a, &b);
       long myresult = add(a,b);
       printf("Sum of numbers = %d\n", myresult);
       return 0;}
```

Listing 1: addTwoNumbers.c Code

```python
def add(a, b):
    return int(a) + int(b)

a, b = input('Enter two numbers!').split()
print('The sum is: {}'.format(add(a,b)))
```

Listing 2: addTwoNumbers.py Code

Simple Functions

Empty Functions

- `pass` is often used in the development process
- Useful, when parts of the code are not ready, yet
- without `pass` one gets `IndentationError: expected an indented block` error

```
def unfinished_function():
    pass


unfinished_function()
```

Listing 3: funktion-12.py Code

Simple Functions

- No parameter
- No return value (`void`)

```
def print_hello():
    print('Hello')


print_hello()
```

Listing 4: funktion-01.py Code

```
Hello
```

Functions with Arguments

- Function with one argument `text`
- Error message, when argument is missing

```
def print_text(text):
    print(text)


print_text('Hello World!')
```

Listing 5: funktion-02.py Code

```
Hello World!
```

Functions with Arguments

- Two arguments, `text` and `count`

```
def print_text_multiple(text, anzahl):
    print(text*count)


print_text_multiple('Hello!', 0)

print_text_multiple('Hello!', 1)

print_text_multiple('Hello!', 3)

print_text_multiple('Hello!', -1)
```

Listing 6: funktion-03.py Code

```

Hello!
Hello!Hello!Hello!
```

Functions with Arguments

- Setting standard values for parameters
- Allows calling the function without parameters

```
def print_text_multiple(text='Hello CGN', count=2):
    print(text*count)


print_text_multiple()

print_text_multiple('Hello!')

print_text_multiple('Hello!',3)
```

Listing 7: funktion-04.py Code

```
Hello CGNHello CGN
Hello!Hello!
Hello!Hello!Hello!
```

Functiona with return values

- Functions can return values for further processing

```
def clone_text(text, count=2):
    return text*count


a = clone_text('Huhu')

print(a)
```

Listing 8: funktion-05.py Code

```
HuhuHuhu
```

Functions with multiple return values

- Functions can have more than one return value
- Function then return tuples, an unmutuable list of values[1]
- Dealing with the tuple is called „Unpacking“
- Remark: Parameter sep in the example is a parameter of the `print()` function

```
def clone_text(text, count=2):
    return count, text*count


i, a = clone_text('Huhu')

print(i, a, sep='>')
```

Listing 9: funktion-06.py Code

```
2>HuhuHuhu
```

[1] more on this later

Functions with variable count of arguments

- Parameter with *: variable count of arguments
- Parameter with **: variable count of key-value arguments

```
def addthem(*args):
    result = 0
    for number in args:
        result += number
    return result


print(addthem(1, 2, 3, 4, 5))
```

Listing 10: funktion-08.py Code

```
def give(**args):
    for key, value in args.items():
        print(key, value, sep=': ')


print(give(Vorname='Uwe', Nachname='Ziegenhagen'))
```

Listing 11: funktion-09.py Code

Flow control

- Input & Output
  - input
  - print
- Branching
  - if else elif
- Loops
  - for
  - while

Branching

- `if <condition>:`

```
def check_length(text):
    if len(text)>8:
        print('Longer than 8 characters!')

check_length('Köln')
check_length('Düsseldorf')
```

Listing 12: if-01.py Code

```
Longer than 8 characters!
```

Branching

- There is no `switch()` in Python
- `if` statements can be used multiple times
- maybe not the most pythonic approach

```
def check_length(text):
    if len(text)>8:
        print(text, 'Longer than 8 characters!', sep=': ')
    if len(text)<=8:
        print(text, 'Shorter or equal to 8 characters!', sep=': '
            )

check_length('Köln')
check_length('Düsseldorf')
```

Listing 13: if-02.py Code

```
Köln: Shorter or equal to 8 characters!
Düsseldorf: Longer than 8 characters!
```

Branching

- more „pythonic“: `else:`

```
def check_length(text):
    if len(text)>8:
        print(text, 'Longer than 8 characters!', sep=': ')
    else:
        print(text, 'Shorter or equal to 8 characters!', sep=': '
            )

check_length('Köln')
check_length('Düsseldorf')
```

Listing 14: if-03.py Code

```
Köln: Shorter or equal to 8 characters!
Düsseldorf: Longer than 8 characters!
```

Branching

- Nesting von `if <condition>: else:` leads to confusing code

```
def check_length(text):
    if len(text)>8:
        print(text, 'Longer than 8 chars!', sep=': ')
    else:
        if len(text)<=5:
            print(text, 'Shorter or equal 5 chars!', sep=': ')
        else:
            print(text, 'Longer than 5, shorter than 8', sep=': ')

check_length('Köln')
check_length('Berlin')
check_length('Düsseldorf')
```

Listing 15: if-04.py Code

```
Köln: Shorter or equal 5 chars!
Berlin: Longer than 5, shorter than 8
Düsseldorf: Longer than 8 chars!
```

Branching

- `if <condition>: else:` can be shortened to `elif:`

```
def check_length(text):
    if len(text)>8:
        print(text, 'Longer than 8 chars!', sep=': ')
    elif len(text)<=5:
        print(text, 'Shorter or equal 5 chars!', sep=': ')
    else:
        print(text, 'Longer than 5, shorter than 8', sep=': ')

check_length('Köln')
check_length('Berlin')
check_length('Düsseldorf')
```

Listing 16: if-05.py Code

```
Köln: Shorter or equal 5 chars!
Berlin: Longer than 5, shorter than 8
Düsseldorf: Longer than 8 chars!
```

Loops

for

- `for` loops iterate over a sequence
- sequence can be a string, a liste, etc.
- `range(start, end, stepsize=1)` creates numerical sequence from `start` until below (!) `end` with step size `stepsize`

```
for char in 'Hallo Welt':
    print(char)

for j in range(1, 10):
    print(j) # 1 bis 9

for j in range(1, 10, 3):
    print(j) # 1, 4 und 7
```

Listing 17: for-01.py Code

Loops

while

- `while` loop runs, until condition is not fulfilled anymore

```
s = 'Hallo Welt'
l = len(s)

while (l>0):
    print(s[l-1])
    l-=1

i = 1
while (i<100):
    i += 1

print(i)
```

Listing 18: while-01.py Code

Loops

break and continue

- break and continue influence loops
- break can e. g. be used to exit a loop

```
s = 'Hallo Welt'
l = len(s)

while (l>0):
    temp = s[l-1]
    if temp == 'W':
        break
    print(temp)
    l-=1
```

Listing 19: while-02.py Code

pandas

pandas

- A Python library for data wrangling and management
- Invented by Wes McKinney during his time at AQR Capital Management
- In his own words: „I tell them that it enables people to analyze and work with data who are not expert computer scientists," he says. „You still have to write code, but it's making the code intuitive and accessible. It helps people move beyond just using Excel for data analysis."[2]

[2] qz.com/1126615/the-story-of-the-most-important-tool-in-data-science/

Das SciPy Framework

`pandas` is just a piece among many

NumPy	Matrices, vectors, algorithms
IPython	Matlab/Mathematica-like environment
Matplotlib	Scientific Plotting, Basis for `seaborn` library
SymPy	Symbolic mathematics
...	etc, etc

We only focus on `pandas`:

```
import pandas as pd
```

Series and DataFrames

- central data structures in `pandas`: `series` and `dataframes`
- quite similar to dataframes in R
- Definition „Series“: a vector with data of the same type and an index
- Definition „Dataframe“: matrix from various series, the series can have different data types haben but they share the same index

Series und DataFrames

Column Index

Row Index

	'var 0'	'var 1'	'var 2'	'var 3'	'var 4'	'var 5'	'var 6'
0	0.2	'USD'	...				
1	0.4	'EUR'	...				
2	0.1	'USD'	...				
3	0.7	'EUR'	...				
4	0.5	'YEN'	...				
5	0.5	'USD'	...				
6	0.0	'AUD'	...				

Manual Creation of pandas Objects

- pandas-objecs *can* be created manually
- normally not used, data is usually loaded from files/databases

```
import pandas as pd

d = pd.DataFrame({'A': ['A0','A1','A2','A3'],
  'Key': ['K0','K1','K2','K4']})
a = pd.Series([1,2,3,4,5,6,7,8,9,10])
b = pd.Series(['A','C','D','B','F','G','I','K','L','P'])
df = pd.concat([a,b], axis=1)
# alternatively
df = pd.DataFrame({'a': a,'b':b})
df = a.to_frame().join(b.to_frame())
df = pd.DataFrame(data=dict(a=a, b=b))
```

Daten einlesen

- various functions to read files

Befehl	Beschreibung
read_pickle	reads Pickle objects
read_table	table-like formats
read_csv	Comma-Separated Values
read_fwf	fixed-width formats
read_clipboard	clipboard
read_excel	Excel-files

other commands for HTML, JSON, HDF5, ...

Reading CSV

- „CSV“: Comma-Separated Value
- CSV is not a unique format
  - Column-separator
  - Decimal-separator
  - Text Encoding
- Specifications: `http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html`

sep	Column-separator
thousands	seperator for thousands
encoding	Encoding
decimal	Decimal-separator
converters	converters={'A': str} for explicit conversion to a format

Reading Excel

- `pd.read_excel()` to read XLSX-files (!)
- Documentation: `http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html`
- Export to Excel using `pd.to_excel()`
- Remarks:
  - Excel-Export is way slower than CSV-Export
  - Export of well-formatted Excel is possible but takes effort
  - One can even control Excel via COM (Common Object Model)

DataFrames bearbeiten

Exploratory Data Analysis

- We use `Northwind` data as an example
- First task after loading: check data consistency

```
customers = pd.read_excel('Northwind.xlsx', sheetname = '
    Customers')

print(len(customers))  # number of rows
print(customers.head()) # first five rows
print(customers.tail()) # last five rows
print(list(customers)) # list of columns
```

Aufgabe

- Read the files `Bestandsdaten.csv`, `Bestandsdaten_Vortag.csv` and `Bewertung.xlsx`
- Check the data using the functions from last slide
- Export the data to HTML, which columns could be used to connect the data sets with eachother?

Pandas Dataframe Operations

Selection and Filtering I

- pandas has advanced methods for selecting, filtering and transforming data
- Select only specific columns

  ```
  df = df[['colA', 'colB']]
  ```

- Select the top two rows (Index starting at 0)

  ```
  df.iloc[:1]
  ```

- Select only those rows where one row > 50

  ```
  df[df['colA'] > 50]
  ```

Pandas Dataframe Operations

Selection and Filtering II

- Select only those rows where value is between two values

```
df[(df['colA'] > 50)| (df['colA'] < 500)]
```

- Select rows that **do not** have a certain value

```
df[~(df['colA'] == 'HelloWorld')]
```

- Select rows, where column b is either value 'A' or 'B'

```
df = df[ (df['b'] == 'A')| (df['b'] == 'I')]
```

- Use alternatively `isin()`

```
df = df[df['b'].isin(['A','I'])]
```

- or the opposite

```
df = df[~df['b'].isin(['A','I'])]
```

- More here: `http://chrisalbon.com/python/pandas_indexing_selecting.html`

Apply functions to pandas columns

```python
import pandas as pd

def capitalizeColumn(text):
    return text.capitalize()

df = pd.DataFrame({'Key': ['K0','K1','K2','K4'],
        'Name': ['anna','bernd','cesar','dana']})

print(df)

df['Nachname'] = df['Name'].apply(capitalizeColumn)

print(df)
```

```
  Key   Name Nachname
0  K0   anna    Anna
1  K1  bernd   Bernd
2  K2  cesar   Cesar
3  K4   dana    Dana
```

Mapping I

- Similar to Excel's `vlookup()` function
- Example: `countries` is a key-value dictionary
- `country` column in the dataframe is used as key in the dictionary, a new column is created

```
import pandas as pd

df = pd.DataFrame({'A': ['A0','A1','A2','A3'],
  'Country': ['DEU','USA','ARE','ESP']})

countries = { 'DEU':'Germany',
            'USA':'United States',
            'ARE':'Arabic Emirates',
            'ESP':'Spain'}

df['Country'] = df['Country'].map(countries)

print(df)
```

Mapping II

- `map()` can also be used to make simple calculations
- keyword here is „lambda“ ⇒ anonymous function

```
import pandas as pd

df = pd.DataFrame({'A': ['A0','A1','A2','A3'],
  'Net': [100, 200, 300, 400]})

df['Gross'] = df['Net'].map(lambda x: x * 1.19)

print(df)
```

Pandas Dataframe Operations

Merge and Join

- `merge()` SQL-like merging of datasets
- useful to combine data
- Supports the following join types:
  - Left
  - Right
  - Inner
  - Full Outer
- `join()` is a special alias for `merge()`, works on the index, not the columns

Pandas Dataframe Operations

Merge and Join

- Standard-command for `merge()`

```
leftDataFrame.merge(rightDataFrame, how='inner',
on=None, left_on=None, right_on=None, left_index=False,
right_index=False, sort=False, suffixes=('_x', '_y'),
copy=True, indicator=False)
```

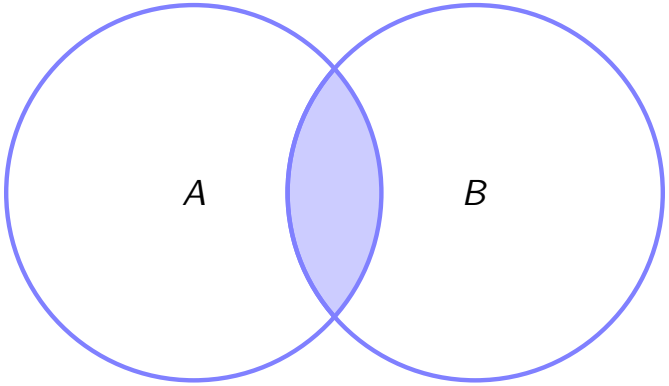
Workflow

1. define other dataset
2. define type of merge
3. define keys for the merger

Merging

Inner Join

- Select all data, that is in A **and** B



left

	A	Key
0	A0	K0
1	A1	K1
2	A2	K2
3	A3	K4

right

	B	Key
0	B0	K0
1	B1	K1
2	B2	K2
3	C3	K5

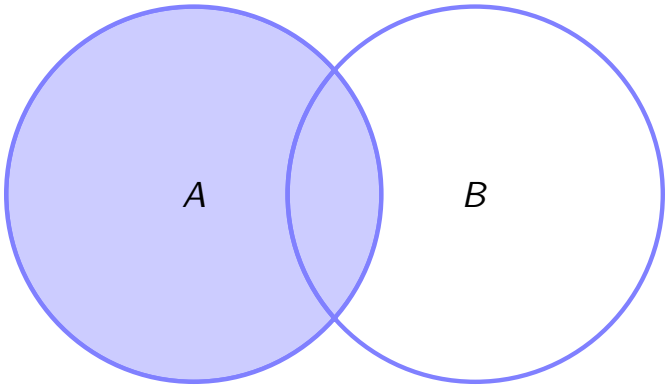
merged

	A	B	Key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2

Merging

Left Join

- Select all data in A, get data from B if available



left

	A	Key
0	A0	K0
1	A1	K1
2	A2	K2
3	A3	K4

right

	B	Key
0	B0	K0
1	B1	K1
2	B2	K2
3	C3	K5

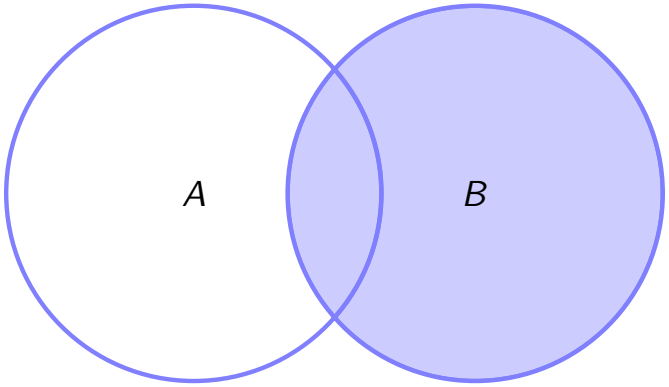
merged

	A	B	Key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	NaN	K4

Merging

Right Join

- Select all data in B, get data from A if available



left

	A	Key
0	A0	K0
1	A1	K1
2	A2	K2
3	A3	K4

right

	B	Key
0	B0	K0
1	B1	K1
2	B2	K2
3	C3	K5

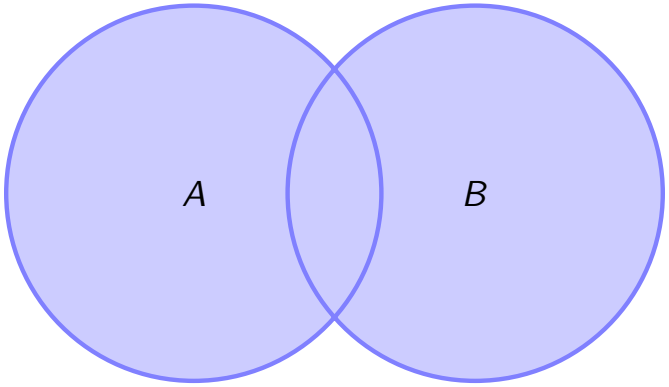
merged

	A	B	Key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	NaN	B3	K5

Merging

Full Outer Join

- Select all data which is in A **or** B



left

	A	Key
0	A0	K0
1	A1	K1
2	A2	K2
3	A3	K4

right

	B	Key
0	B0	K0
1	B1	K1
2	B2	K2
3	C3	K5

merged

	A	B	Key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	NaN	K4
4	NaN	B3	K5

Loop through DataFrames

```python
import pandas as pd

df = pd.DataFrame({'Key': ['K0','K1','K2','K4'],
        'Name': ['Anna','Bernd','Cesar','Dana']})

for index, row in df.iterrows():
    print(row['Key'], 'belongs to', row['Name'])
```

```
K0 belongs to Anna
K1 belongs to Bernd
K2 belongs to Cesar
K4 belongs to Dana
```

Links

Links

- https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
- https://pandas.pydata.org
-
-
-
-