Pandas

Data Science 2 / Data & AI 3

Karel de Grote Hogeschool

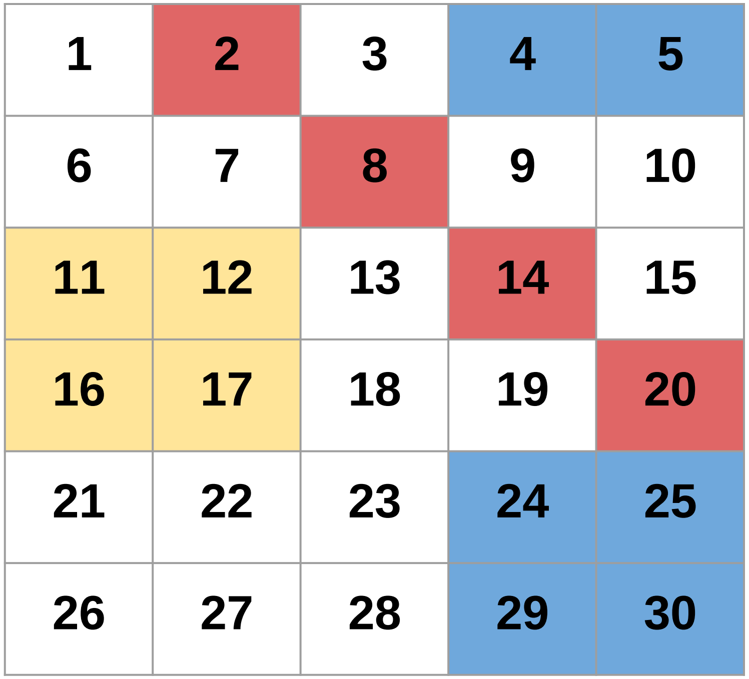
Revision

Revision - Indexing

- What is the proper indexing to retrieve the the values in the yellow squares?

- The blue squares?

- The red squares?



1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

Revision

- What is the type of this numpy array?

```
X= np.array(
    [[1,2,3],
    [4,"5",6],
    [7,8,9]
   ])
```

Revision

- How to replace items that satisfy a condition without affecting the original array?

The input is: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

The expected output is: array([0, -1, 2, -1, 4, -1, 6, -1, 8, -1])

Agenda

1. Introduction to Pandas
2. Indexing and Selection
3. Operations and Missing Values
4. Merge and Join
5. Aggregation and Grouping
6. Working with Strings
7. Working with Time Series
8. Reading files

Cursus/
Boek

canvas Oefeningen

1

Introduction to Pandas

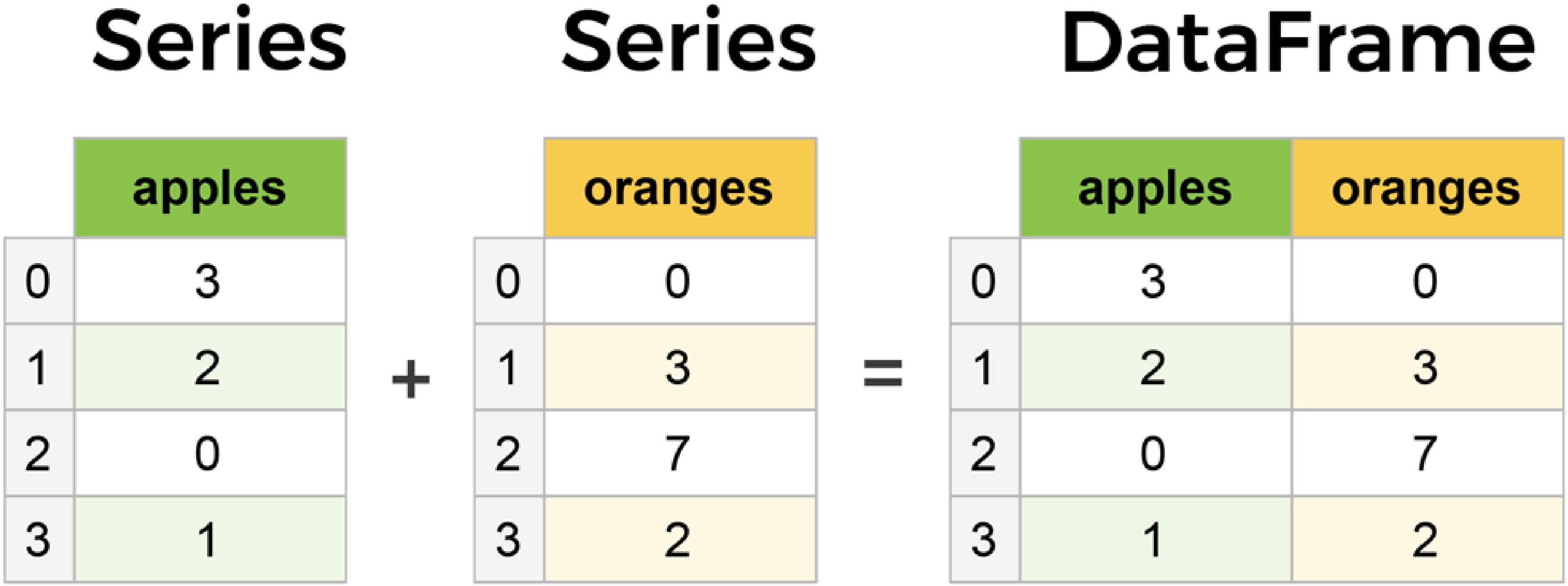
What is Pandas

Python library with flexible data structures developed for Data Scientists

DataFrame

Series

Data Structures are build on Numpy arrays



Series

	apples
0	3
1	2
2	0
3	1

+

Series

	oranges
0	0
1	3
2	7
3	2

=

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

source: https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/

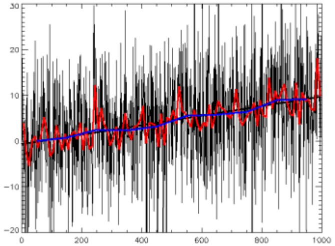
What is Pandas

Importing exporting and processing multiple data sources

Uniform handling missing data

N.A.

Explicitly defined indexes enabling advanced indexing, slicing and subsetting



Time series functionality

Advanced data manipulation

- GroupBy
- Joining
- ...

Pandas Series

Series as generalized NumPy array

- Numpy array: implicitly defined integer index
- Pandas Series: explicitly defined index

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
data[5]                                                        # 0.5
```

Series as specialized dictionary

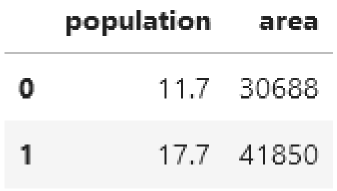
- Python dictionary: values can have different types
- Pandas Series: all values have the same type (efficiency!)

```
population = pd.Series({'be': 10, 'nl': 8})
data['be'] ]                                                   # 10
```

Pandas Dataframes

Dataframe as generalized 2D NumPy array

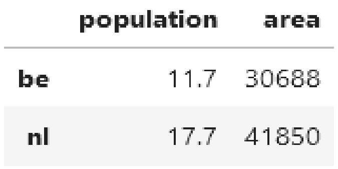
```
countries = pd.DataFrame( [ {'population': 11.7, 'area': 30688},
                            {'population': 17.7, 'area': 41850} ] )
countries
```



	population	area
0	11.7	30688
1	17.7	41850

Series as specialized dictionary

```
population = pd.Series({'be': 11.7, 'nl': 17.7})
area = pd.Series({'be': 30688, 'nl': 41850})
countries = pd.DataFrame({'population': population, 'area': area})
countries['area']                                          # or countries.area
```



	population	area
be	11.7	30688
nl	17.7	41850

Notebook and Exercise time!

Notebook

See 03.01-Introducing-Pandas-Objects.ipynb

Exercise time!

See 03.01_EX.ipynb

pandas

Indexing and Selection

2

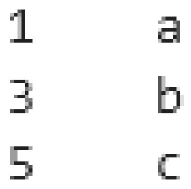
Indexing and Selection

1. Data Selection in Series
   - Series as dictionary
   - Series as one-dimensional array
   - Indexers: loc, iloc, and ix
     - Avoid *ix* because it is no longer available in modern pandas versions

2. Data Selection in DataFrame
   - DataFrame as a dictionary
   - DataFrame as two-dimensional array
   - Additional indexing conventions

Indexing and Selection - Series

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
```

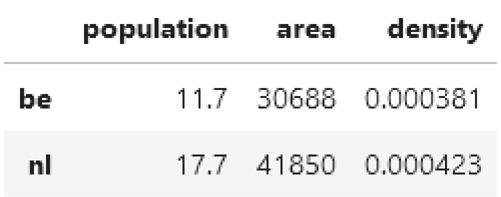


```
1    a
3    b
5    c
```

```
# explicit index: .loc
data.loc[1]                             # 'a’
# slicing
data.loc[1:3]                           # 1 a
                                        # 3 b, explicit index: final index is included

# implicit index: .iloc
data.iloc[1]                            # ‘b’
# slicing
data.iloc[1:3]                          # 3 b
                                        # 5 c, implicit index: final index is excluded

# masking and fancy indexing
data[(data == 'a') | (data == 'b’)]
data.loc[[1,3]]
```

Indexing and Selection - Dataframe

```
data= pd.DataFrame([ {'population': 11.7, 'area': 30688},
                     {'population': 17.7, 'area': 41850}], index=['be', 'nl'])

data['density'] = data['pop'] / data['area']
```



	population	area	density
be	11.7	30688	0.000381
nl	17.7	41850	0.000423

```
# implicit index: .iloc
data.iloc[:1, :1]              # implicit index: final index is excluded -> 1x1

# explicit index: .loc
data.loc[: 'nl ', : 'area']    # explicit index: final index is included -> 2x2
# with masking and fancy indexing
data.loc[data.population>15, ['area', 'density']]
```

Notebook and Exercise time!

Notebook

See 03.02-Data-Indexing-and-Selection

Exercise time!

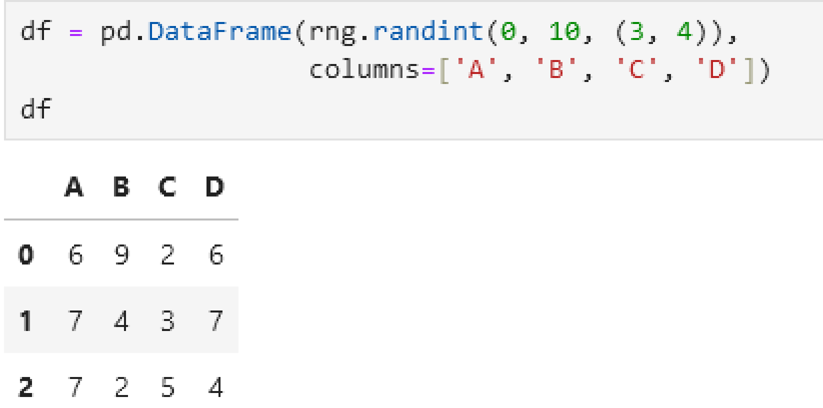
See 03.02_EX.ipynb

Operations and Missing Values in Pandas

3

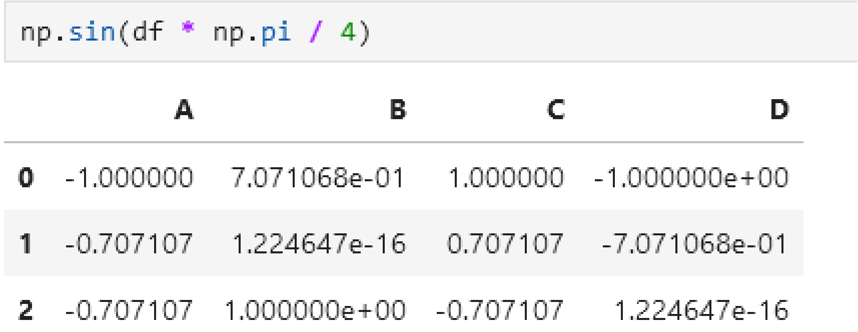
Operating on Data in Pandas

Operations in Pandas
Ufuncs: Index Preservation
Ufuncs: Index Alignment
Operations Between DataFrame and Series



```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4



```
np.sin(df * np.pi / 4)
```

	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

Missing values

1. Handling Missing Data

2. Trade-Offs in Missing Data Conventions*

3. Missing Data in Pandas*
   - `None`: Pythonic missing data*
   - `NaN`: Missing numerical data*
   - NaN and None in Pandas*

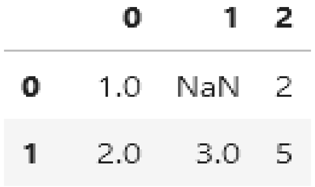
4. Operating on Null Values
   - Detecting null values
   - Dropping null values
   - Filling null values

* Reading for context suffices

Missing Values

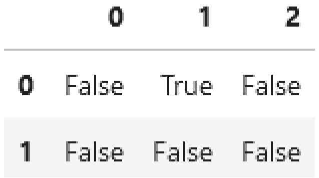
Pandas treats None and NaN as essentially interchangeable for indicating missing or null values

```
df = pd.DataFrame([[1,    np.nan, 2],
                   [2,    3,         5]])
```



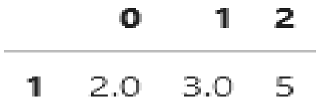
	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5

```
# detecting null values
df.isnull()
df.notnull()
```



	0	1	2
0	False	True	False
1	False	False	False

```
# dropping null values
df.dropna()                               # drops rows
df.dropna(axis='columns', thresh=3)       # drops columns, with min 3 Nas
```



	0	1	2
1	2.0	3.0	5

```
# filling null values
df.fillna(0)                              # fill with Nas with 0
```

Notebook and Exercise time!

Notebook

See 03.04-Missing-Values.ipynb

See 03.04-Missing-Values.ipynb

Exercise time!

See 03.03_EX.ipynb

See 03.04-EX.ipynb

Merge and Join

4

Merge and Join

1. Combining Datasets: Merge and Join

2. Relational Algebra

3. Categories of Joins
   - One-to-one joins
   - Many-to-one joins
   - Many-to-many joins

4. Specification of the Merge Key
   - The `on` keyword
   - The `left_on` and `right_on` keywords
   - The `left_index` and `right_index` keywords

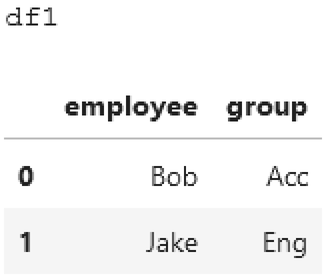
5. Specifying Set Arithmetic for Joins

6. Overlapping Column Names: The `suffixes` Keyword

7. Example: US States Data

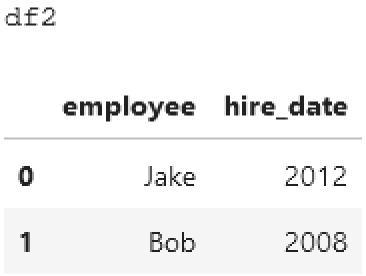
Merge and Join

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake'],
                    'group': ['Acc', 'Eng',]})
df2 = pd.DataFrame({'employee': [ 'Jake', 'Bob'],
                    'hire_date': [ 2012, 2008]})
```



df1

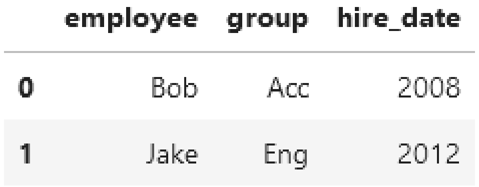
	employee	group
0	Bob	Acc
1	Jake	Eng



df2

	employee	hire_date
0	Jake	2012
1	Bob	2008

```
# merge detects common column
pd.merge(df1, df2)
# can merge one-to-one, one-to-many, many-to-many
```



	employee	group	hire_date
0	Bob	Acc	2008
1	Jake	Eng	2012

```
# merge with different column names
pd.merge(df1, df3, left_on="employee", right_on="name“

# merge on index
df1a.join(df2a)          # same as pd.merge(df1a, df2a, left_index=True, right_index=True)

# merge on index and column
pd.merge(df1a, df3, left_index=True, right_on='name')
```

Merge and Join

```
# default is 'inner’ join
# 'outer', 'left', and 'right' joins
pd.merge(df6, df7, how='outer’)”

# overlapping column names
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

Notebook and Exercise time!

Notebook

See 03.07-Merge-and-Join.ipynb

Exercise time!

See 03.07-EX.ipynb

Aggregation and Grouping

5

Aggregation and Grouping

1. Aggregation and Grouping

2. Planets Data

3. Simple Aggregation in Pandas

4. GroupBy:
   - Split, apply, combine
   - The GroupBy object
     - Column indexing
     - Iteration over groups
     - Dispatch methods

Aggregation and Grouping

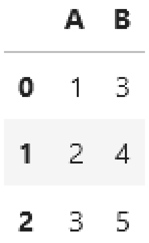
4. GroupBy: Split, Apply, Combine
   - Aggregate, filter, transform, apply
     - Aggregation
     - Filtering
     - Transformation
     - The apply() method

- Specifying the split key
  - A list, array, series, or index providing the grouping keys
  - A dictionary or series mapping index to group
  - Any Python function
  - A list of valid keys

- Grouping example

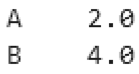
Simple Aggregation

```
df = pd.DataFrame({'A': [1, 2, 3],
                   'B': [3, 4, 5]})
```



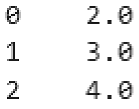
	A	B
0	1	3
1	2	4
2	3	5

```
df.mean()
```



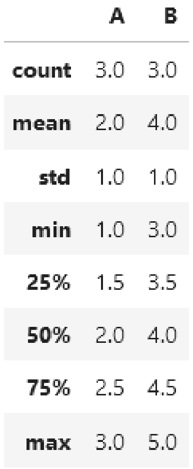
```
A     2.0
B     4.0
```

```
df.mean(axis='columns’)
```

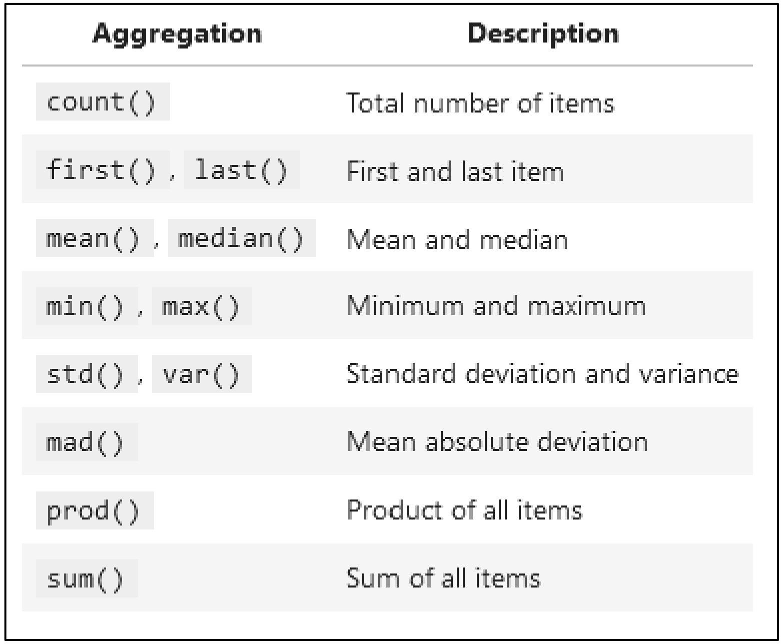


```
0     2.0
1     3.0
2     4.0
```

```
df.describe()
```



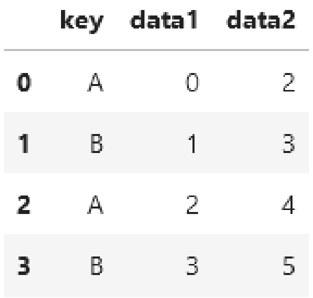
	A	B
count	3.0	3.0
mean	2.0	4.0
std	1.0	1.0
min	1.0	3.0
25%	1.5	3.5
50%	2.0	4.0
75%	2.5	4.5
max	3.0	5.0



Aggregation	Description
`count()`	Total number of items
`first()` , `last()`	First and last item
`mean()` , `median()`	Mean and median
`min()` , `max()`	Minimum and maximum
`std()` , `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

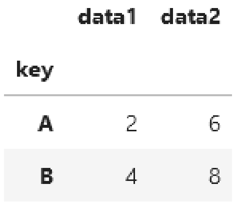
GroupBy

```
df = pd.DataFrame({'A': [1, 2, 3],
                   'B': [3, 4, 5]})
```

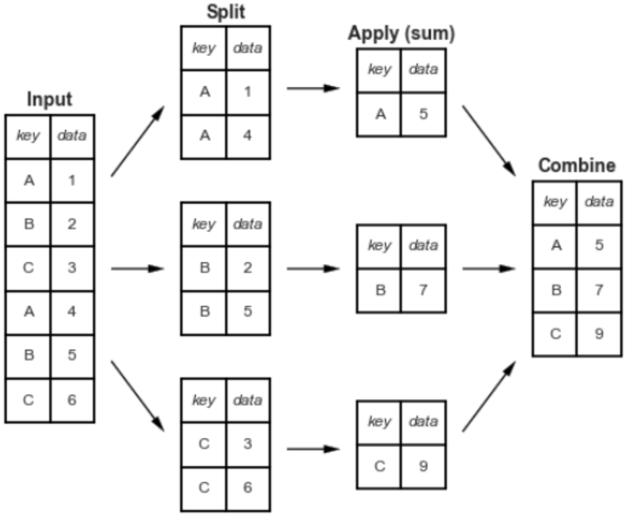


	key	data1	data2
0	A	0	2
1	B	1	3
2	A	2	4
3	B	3	5

```
df.groupby('key').sum()
```



	data1	data2
key		
A	2	6
B	4	8



Input

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

Split

key	data
A	1
A	4

key	data
B	2
B	5

key	data
C	3
C	6

Apply (sum)

key	data
A	5

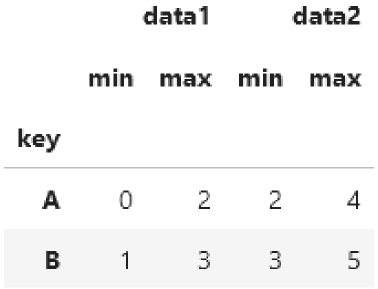
key	data
B	7

key	data
C	9

Combine

key	data
A	5
B	7
C	9

```
df.groupby('key').aggregate(['min', 'max’])
```



	data1		data2	
	min	**max**	**min**	**max**
key				
A	0	2	2	4
B	1	3	3	5

```
df.groupby('key').apply(your_own_function)
```

Notebook and Exercise time!

Notebook

See 03.08-Aggregation-and-Grouping.ipynb

Exercise time!

See …

6

Working with Strings

Working with strings

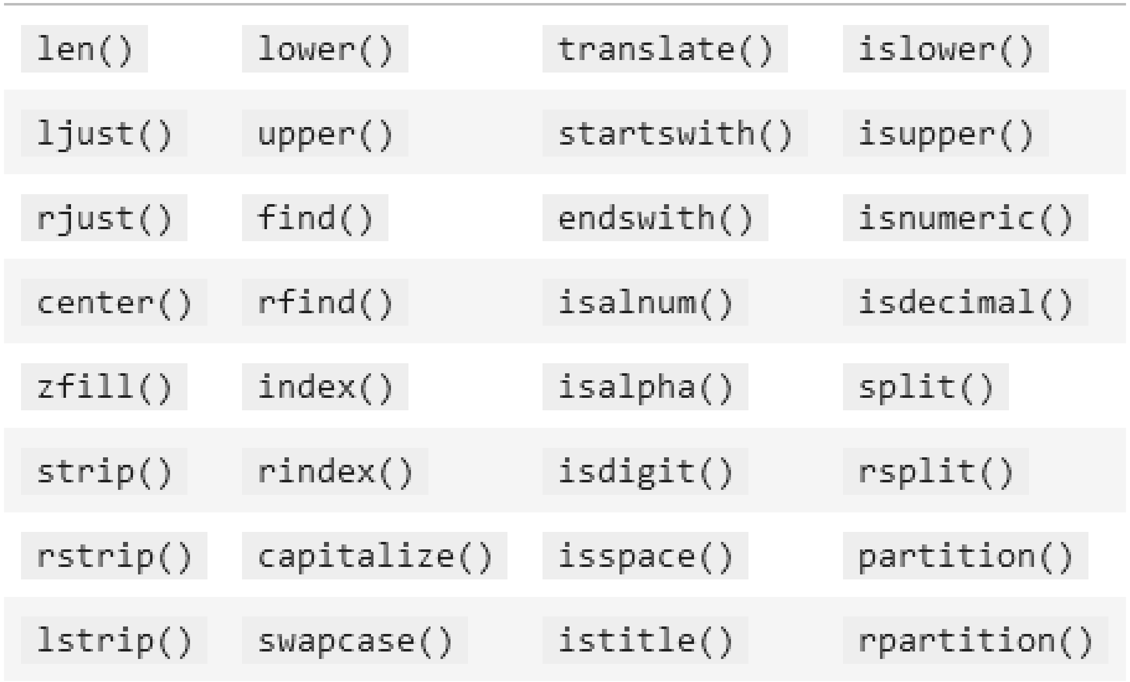
1. Vectorized String Operations

2. Introducing Pandas String Operations

3. Tables of Pandas String Methods
    - Methods similar to Python string methods
    - Methods using regular expressions
    - Miscellaneous methods
        - Vectorized item access and slicing
        - Indicator variables

GroupBy

String methods



`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Notebook and Exercise time!

Notebook

See 03.10-Working-With-Strings.ipynb

Exercise time!

See 3.10_EX_strings.ipynb

7

Reading Files

Reading files

1. Reading Data

2. Reading CSV files and working with a dataframe

3. Categorical Variables

Reading files

```
# reading csv file
data = pd.read_csv('file_name')
data = pd.read_csv('file_name', sep=';')                        # separator is ;
data = pd.read_csv('file_name', sep='; ', decimal=',')          # decimal point is ,
data = pd.read_csv('file_name', names=['n1', 'n2'])             # if header is not in file

# categorical variables
bloodtype = pd.Categorical(values, categories=['O-','O+','B-','B+','A-','A+','AB-','AB+’])

# define columns as categorical
laptops = pd.read_csv('laptops.csv', dtype={'cpu': 'category', 'brand': 'category'})
```

Notebook and Exercise time!

Notebook

See 03.XTR_ReadingFiles.ipynb

Exercise time!

See …