Python for Data Analysis

Data Cleaning and Preparation (Week 7)

Atefeh Khazaei

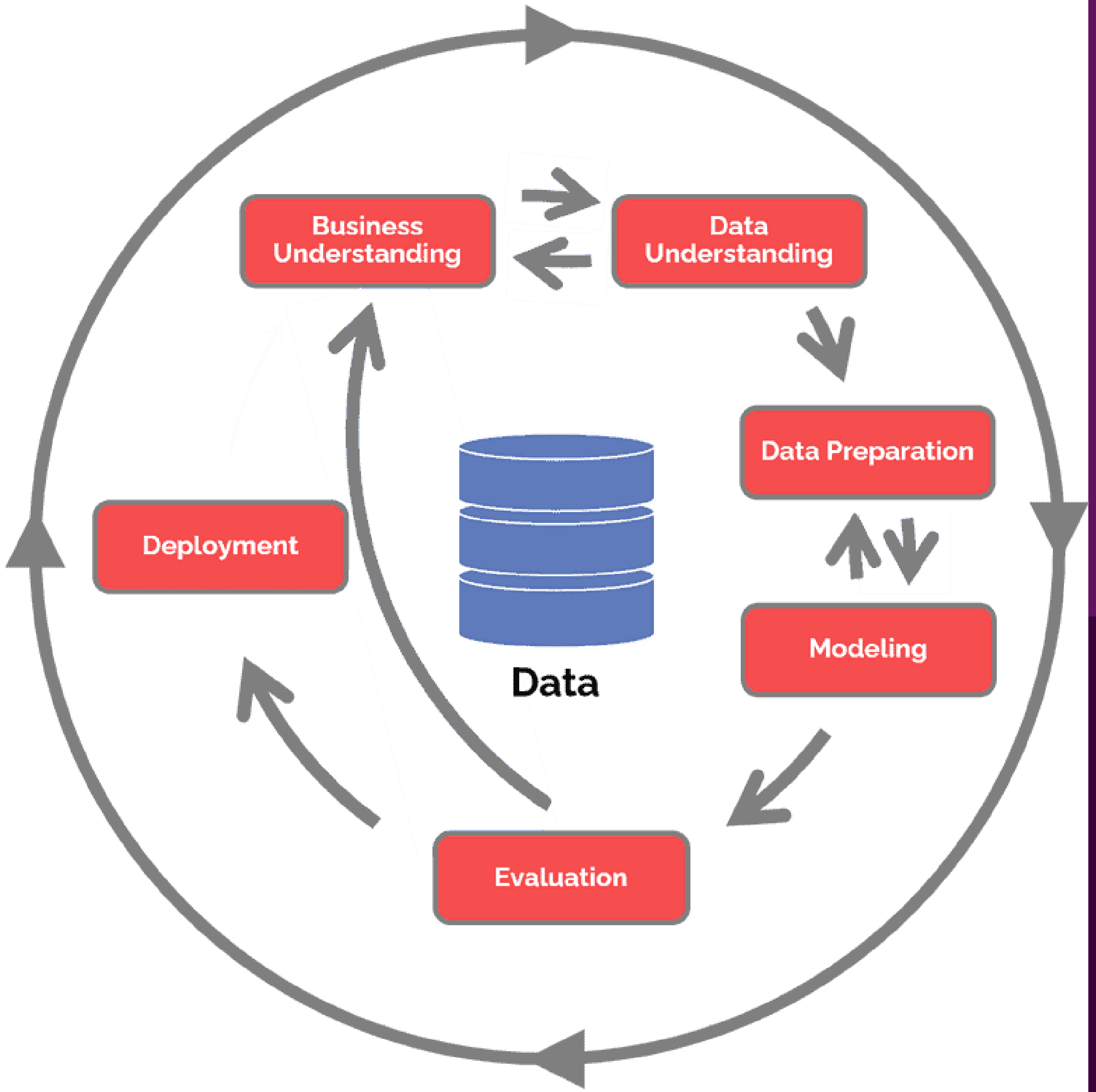
atefeh.khazaei@port.ac.uk

Flashback of the First Week

- The aim of this unit is to improve your skills on some data analysis operations using Python programming.
- Data analysis definition
- Data analysis job vacancies

CRISP-DM

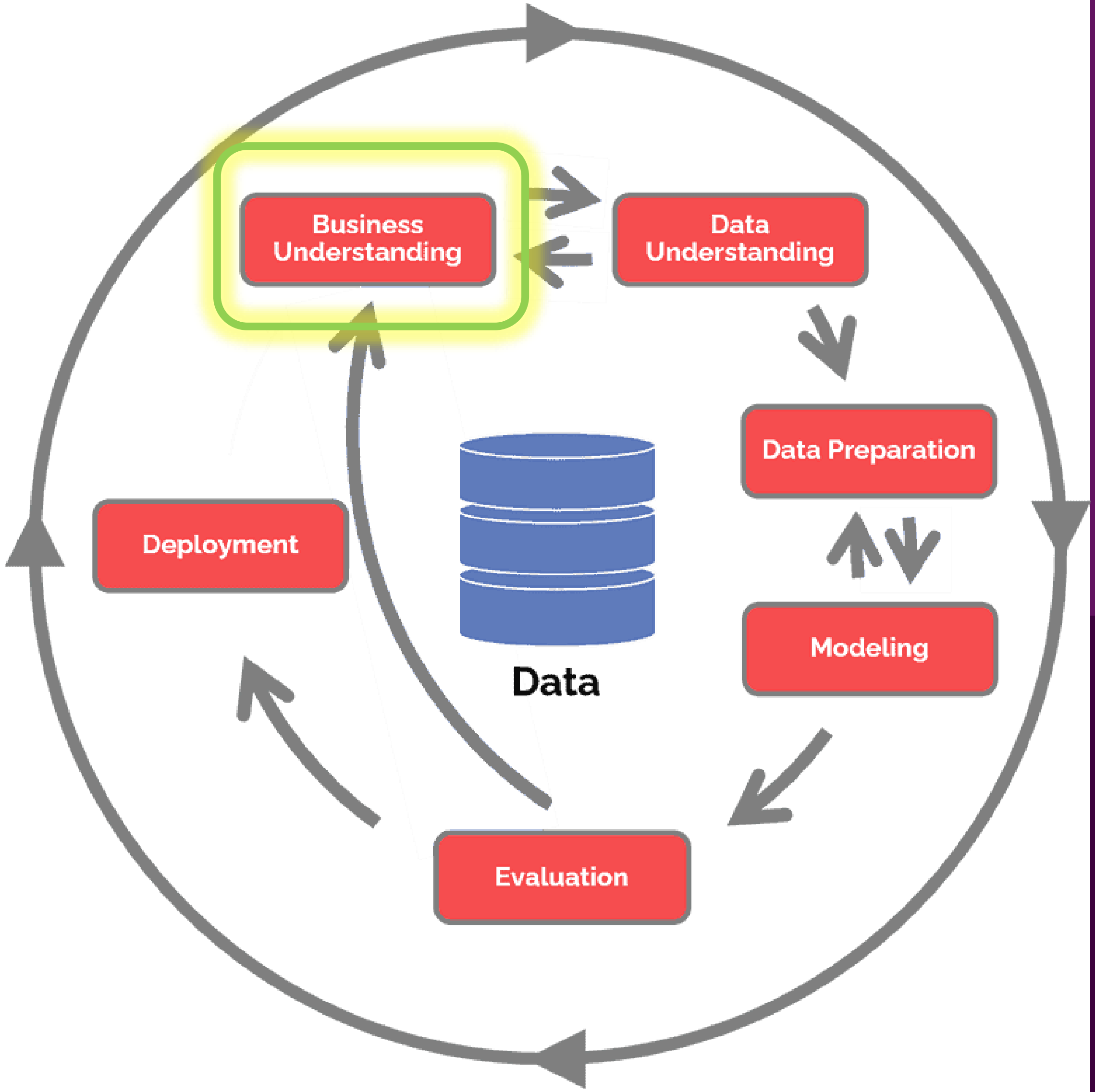
- The **CR**oss **I**ndustry **S**tandard **P**rocess for **D**ata **M**ining (CRISP-DM) is a process model with six phases that naturally describes the data science life cycle.



- https://www.datascience-pm.com/crisp-dm-2/

CRISP-DM (cont.)
Data Understanding

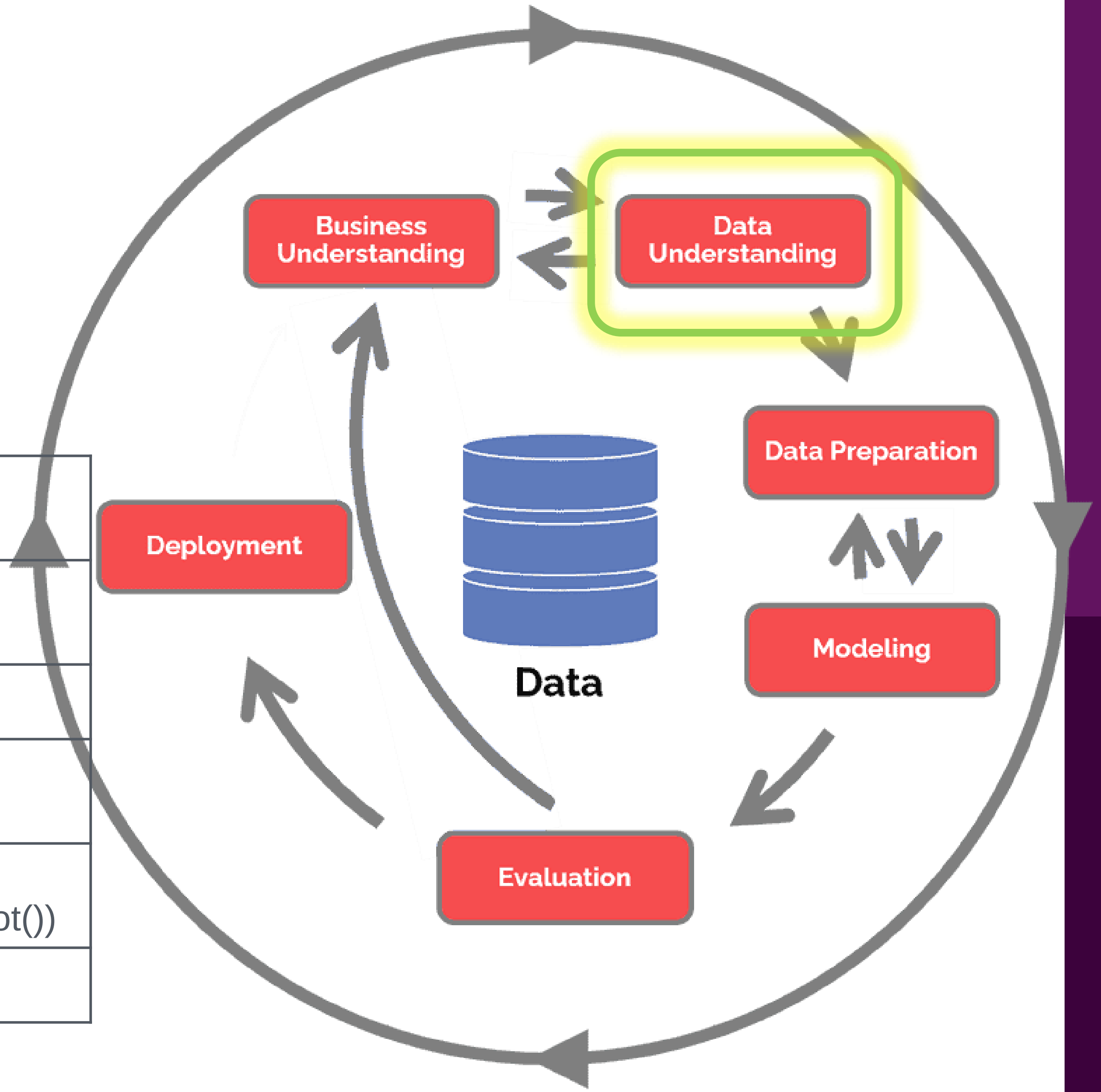
- ❑ **Business understanding**
  - ❑ What does the business need?



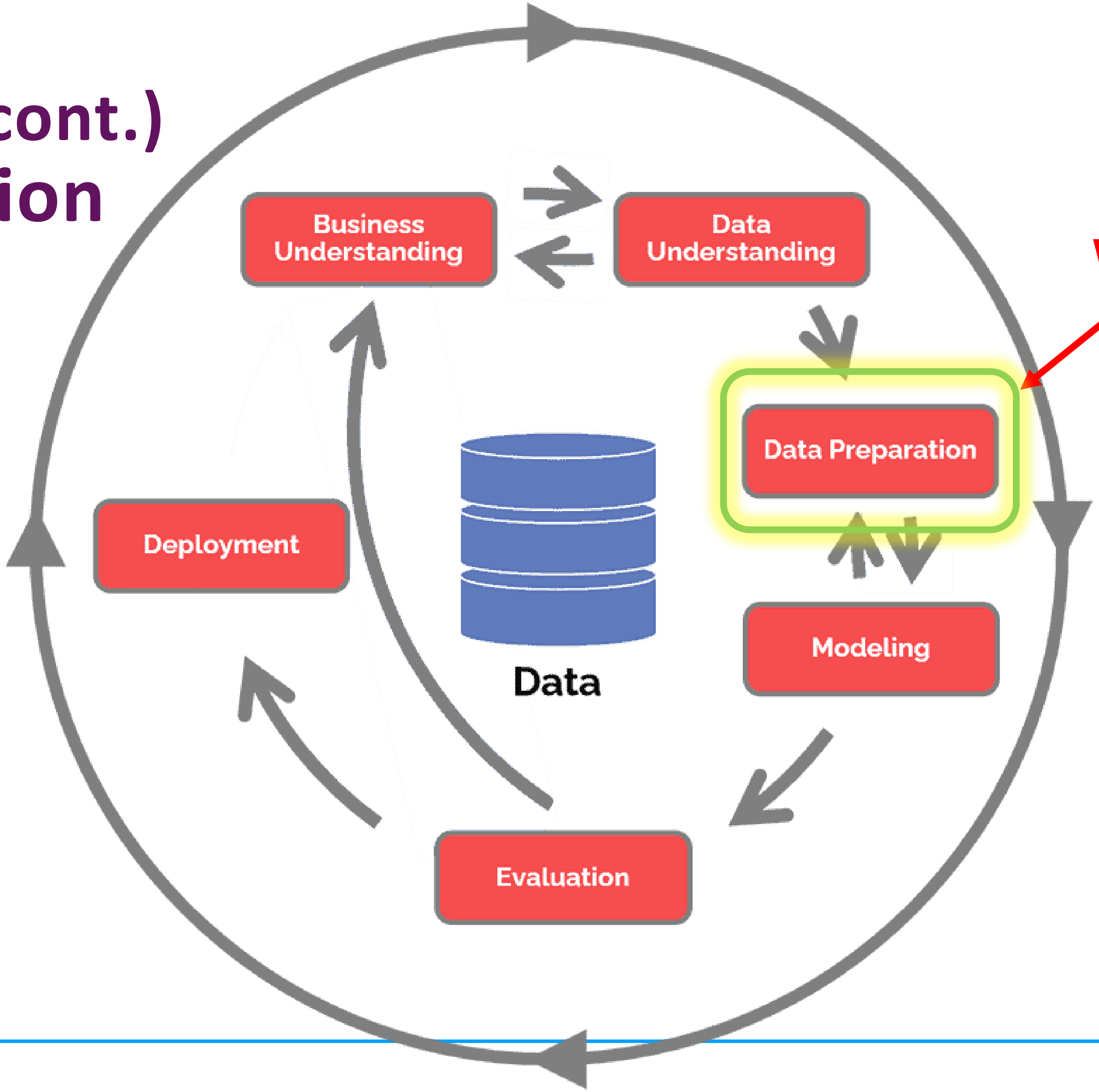
CRISP-DM (cont.)
Data Understanding

- ❑ What data do we have? Is it clean?

Informative functions (e.g. type(), head())	Loading data functions (e.g. read_csv(), loadtxt())
Python data structures (tuple, list, dict)	Save data functions (e.g. to_csv(), save(), savetxt())
NumPy arrays (n-dimentional arrays)	Filtering in pandas
Pandas data structures (Series, DataFrame)	Mergeing DataFrames (merge() and concat() functions)
Useful functions like isnull() and notnull()	Plotting some charts (e.g. imshow(), heatmap(), pairplot())
Create and del DataFrame columns	**etc.**



CRISP-DM (cont.)
Data Preparation



We are here.

What we will learn this week?

- ❑ Handling Missing Data
- ❑ Data Transformation
- ❑ String Manipulation

Data Preparation

- A significant amount of time is spent on data preparation:
  - Loading,
  - Cleaning,
  - Transforming,
  - Rearranging.
- Such tasks are often reported to take up 80% or more of an analyst's time.
- Sometimes the way that data is stored in files or databases is not in the right format for a particular task.

Handling Missing Data

- One of the goals of pandas is to make working with missing data as painless as possible.
- A list of some functions related to missing data handling:

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

Handling Missing Data (cont.)
Filtering Out Missing Data

- There are a few ways to filter out missing data.
- On a Series:

```
import pandas as pd
from numpy import nan as NA
data = pd.Series([1, NA, 3.5, NA, 7])
```

```
data.dropna()
```

```
0    1.0
2    3.5
4    7.0
dtype: float64
```

```
data[data.notnull()]
```

```
0    1.0
2    3.5
4    7.0
dtype: float64
```

Handling Missing Data (cont.)
Filtering Out Missing Data

- With DataFrame objects, things are a bit more complex.
- You may want to drop rows or columns that are all NA or only those containing any NAs.
- **dropna** by default drops any row containing a missing value.

```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                     [NA, NA, NA], [NA, 6.5, 3.]])
cleaned = data.dropna()
```

```
data
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
cleaned
```

	0	1	2
0	1.0	6.5	3.0

Handling Missing Data (cont.)
Filtering Out Missing Data

```
data.dropna(how='all')
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

- To drop columns in the same way, pass axis=1:

```
data[4] = NA
data
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
data.dropna(axis=1, how='all')
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Handling Missing Data (cont.)
Filtering Out Missing Data

- Suppose you want to keep only rows containing a certain number of observations.
- You can indicate this with the **thresh** argument:

```
df
```

	0	1	2
0	2.042002	NaN	NaN
1	0.060985	NaN	NaN
2	0.817123	NaN	-0.505980
3	-0.036908	NaN	1.245542
4	0.159401	0.346533	-0.477856
5	-0.250375	0.357300	0.431367
6	1.036217	-0.736127	-0.029084

```
df.dropna(thresh=2)
```

	0	1	2
2	0.817123	NaN	-0.505980
3	-0.036908	NaN	1.245542
4	0.159401	0.346533	-0.477856
5	-0.250375	0.357300	0.431367
6	1.036217	-0.736127	-0.029084

Handling Missing Data (cont.)
Filling In Missing Data

- Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways.

```
df
```

	0	1	2
0	2.042002	NaN	NaN
1	0.060985	NaN	NaN
2	0.817123	NaN	-0.505980
3	-0.036908	NaN	1.245542
4	0.159401	0.346533	-0.477856
5	-0.250375	0.357300	0.431367
6	1.036217	-0.736127	-0.029084

```
df.fillna(0)
```

	0	1	2
0	2.042002	0.000000	0.000000
1	0.060985	0.000000	0.000000
2	0.817123	0.000000	-0.505980
3	-0.036908	0.000000	1.245542
4	0.159401	0.346533	-0.477856
5	-0.250375	0.357300	0.431367
6	1.036217	-0.736127	-0.029084

```
df.fillna({1: 0.5, 2: 0})
```

	0	1	2
0	2.042002	0.500000	0.000000
1	0.060985	0.500000	0.000000
2	0.817123	0.500000	-0.505980
3	-0.036908	0.500000	1.245542
4	0.159401	0.346533	-0.477856
5	-0.250375	0.357300	0.431367
6	1.036217	-0.736127	-0.029084

Handling Missing Data (cont.)
Filling In Missing Data

- With **fillna** you can do lots of other things with a little creativity.

	0	1	2
0	-0.042972	NaN	NaN
1	2.262982	NaN	NaN
2	-0.583276	NaN	0.875014
3	-0.218554	NaN	0.586863
4	0.910501	1.647859	-0.364562
5	-0.869454	-1.136806	-0.757899
6	-1.018473	-0.314309	-0.369785

```
df[2] = df[2].fillna(df[2].mean())
df
```

	0	1	2
0	-0.042972	NaN	-0.006074
1	2.262982	NaN	-0.006074
2	-0.583276	NaN	0.875014
3	-0.218554	NaN	0.586863
4	0.910501	1.647859	-0.364562
5	-0.869454	-1.136806	-0.757899
6	-1.018473	-0.314309	-0.369785

fillna function arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

Data Transformation
Removing Duplicates

- Duplicate rows may be found in a DataFrame for any reasons.
- The DataFrame method duplicated returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not.

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

```
data.duplicated()
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

```
data.drop_duplicates()
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

```
data.drop_duplicates(['k1'])
```

	k1	k2
0	one	1
1	two	1

Data Transformation (cont.)
Replacing Values

- Filling in missing data with the fillna method is a special case of more general value replacement.
- **replace** provides a simpler and more flexible way to do so.

```
import numpy as np
data = pd.Series([1., -999., 2., -999., -1000., 3.])
# The -999 values might be sentinel values for missing data.
data.replace(-999, np.nan)
```

```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

Data Transformation (cont.)
Replacing Values

```
import numpy as np
data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

```
data.replace([-999, -1000], np.nan)
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

```
data.replace([-999, -1000], [np.nan, 0])
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

```
data.replace({-999: np.nan, -1000: 0})
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Data Transformation (cont.)

Discretization and Binning

- Continuous data is often discretized or otherwise separated into “bins” for analysis.

```python
import pandas as pd
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

```python
cats.codes
```

```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```python
cats.categories
```

```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')
```

```python
pd.value_counts(cats)
```

```
(18, 25]     5
(25, 35]     3
(35, 60]     3
(60, 100]    1
dtype: int64
```

Data Transformation (cont.)
Detecting and Filtering Outliers

- Filtering or transforming outliers is largely a matter of applying array operations.
- **Outliers**, being the most extreme observations, may include the sample maximum or sample minimum, or both, depending on whether they are extremely high or low.
- There are lots of methods to detect outliers; but at the moment we are going to suppose that you know which samples are outliers and you want to filter them.

Data Transformation (cont.)
Detecting and Filtering Outliers

```python
import pandas as pd
import numpy as np
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.020221	0.036088	0.058457	-0.030678
std	0.997483	0.987718	0.970755	1.001037
min	-3.234297	-3.113059	-2.943798	-3.736771
25%	-0.722087	-0.604840	-0.603155	-0.711242
50%	-0.008312	0.036988	0.057173	-0.041801
75%	0.640483	0.726219	0.728000	0.647581
max	3.342985	2.825956	3.290140	2.637964

- If you know a threshold value to detect outliers for one column…

```python
# To find values in one of the columns
# exceeding 3 in absolute value.
col = data[3]
col[np.abs(col) > 3]
```

```
85     -3.525415
154    -3.139720
315    -3.736771
611    -3.118009
731    -3.571038
991    -3.136639
Name: 3, dtype: float64
```

Data Transformation (cont.)
Detecting and Filtering Outliers

- If you know a threshold value to detect outliers for all rows…

```
# To select all rows having a value exceeding 3 or -3
data[(np.abs(data) > 3).any(1)]
```

	0	1	2	3
85	1.413356	0.682567	1.084291	-3.525415
154	1.646958	-0.343996	1.583803	-3.139720
158	-3.234297	-0.812317	-0.844487	2.366868
277	-2.156800	0.172825	3.290140	-0.946705
315	-0.472912	-1.048041	-1.025284	-3.736771
465	3.034796	0.823151	0.276075	0.462952
532	-0.558145	-3.113059	0.160536	1.580211
611	0.326295	0.502499	-0.418180	-3.118009
731	0.978080	-0.613736	0.973372	-3.571038
867	0.417708	-3.076463	0.275879	0.732705
921	3.342985	-0.324588	-0.166651	1.640342
991	-0.482081	0.936509	-0.007169	-3.136639

Data Transformation (cont.)
Detecting and Filtering Outliers

- After outlier detection, you can remove OR modify these samples.
  - An example for modification:

```
# Values can be set based on these criteria.
# Here is code to cap values outside the interval -3 to 3:
data[np.abs(data) > 3] = np.sign(data) * 3
data.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.020365	0.036278	0.058166	-0.028450
std	0.995549	0.987125	0.969832	0.993796
min	-3.000000	-3.000000	-2.943798	-3.000000
25%	-0.722087	-0.604840	-0.603155	-0.711242
50%	-0.008312	0.036988	0.057173	-0.041801
75%	0.640483	0.726219	0.728000	0.647581
max	3.000000	2.825956	3.000000	2.637964

Data Transformation (cont.)
Computing Indicator/Dummy Variables

- Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a “dummy” or “indicator” matrix.
- If a column in a DataFrame has k distinct values, you would derive a matrix or DataFrame with k columns containing all 1s and 0s.
- pandas has a get_dummies function for doing this.

Data Transformation (cont.)
Computing Indicator/Dummy Variables

```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                   'data1': range(6)})
df
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
pd.get_dummies(df['key'])
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

Data Transformation (cont.)
Computing Indicator/Dummy Variables

```
dummies = pd.get_dummies(df['key'], prefix='key')
df_with_dummy = df[['data1']].join(dummies)
df_with_dummy
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

String Manipulation
String Object Methods

- Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing.
- Most text operations are made simple with the string object's built-in methods.
  - Example: a comma-separated string can be broken into pieces with **split**.
  - split is often combined with **strip** to trim whitespace.

```
val = 'a,b, guido'
val.split(',')
```

```
['a', 'b', ' guido']
```

```
pieces = [x.strip() for x in val.split(',')]
pieces
```

```
['a', 'b', 'guido']
```

String Manipulation (cont.)
String Object Methods

- Some Example:

```python
val = 'a,b, guido'
'guido' in val
```

```
True
```

```python
val.index(',')
```

```
1
```

```python
val.find(':')
```

```
-1
```

```python
val.count(',')
```

```
2
```

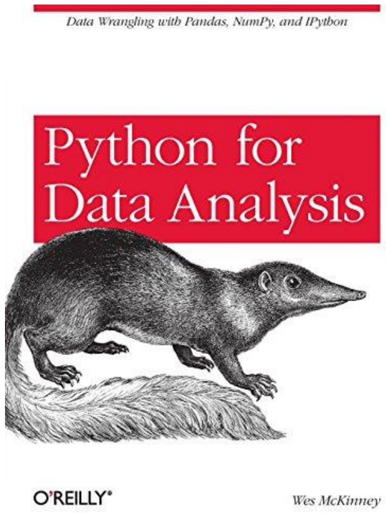
```python
val.replace(',', '::')
```

```
'a::b:: guido'
```

String Manipulation (cont.)
String Object Methods

Argument	Description
`count`	Return the number of non-overlapping occurrences of substring in the string.
`endswith`	Returns `True` if string ends with suffix.
`startswith`	Returns `True` if string starts with prefix.
`join`	Use string as delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string; raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
`replace`	Replace occurrences of string with another string.
`strip`, `rstrip`, `lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
`ljust`, `rjust`	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

References & More Resources



- References:
  - McKinney, Wes. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

- More Resources:
  - Python Data Analysis on Linkedin Learning: https://www.linkedin.com/learning/python-data-analysis-2
  - Learning Python on Linkedin Learning https://www.linkedin.com/learning/learning-python
    - To use Linkedin Learning, you can log in with your university account: https://myport.port.ac.uk/study-skills/linkedin-learning

COURSE

Python Data Analysis

By: Michele Vallisneri

COURSE

Learning Python

By: Joe Marini

Practical Session

- Please read the practical sheet (Week07_Practicals.pdf) and do the exercise.