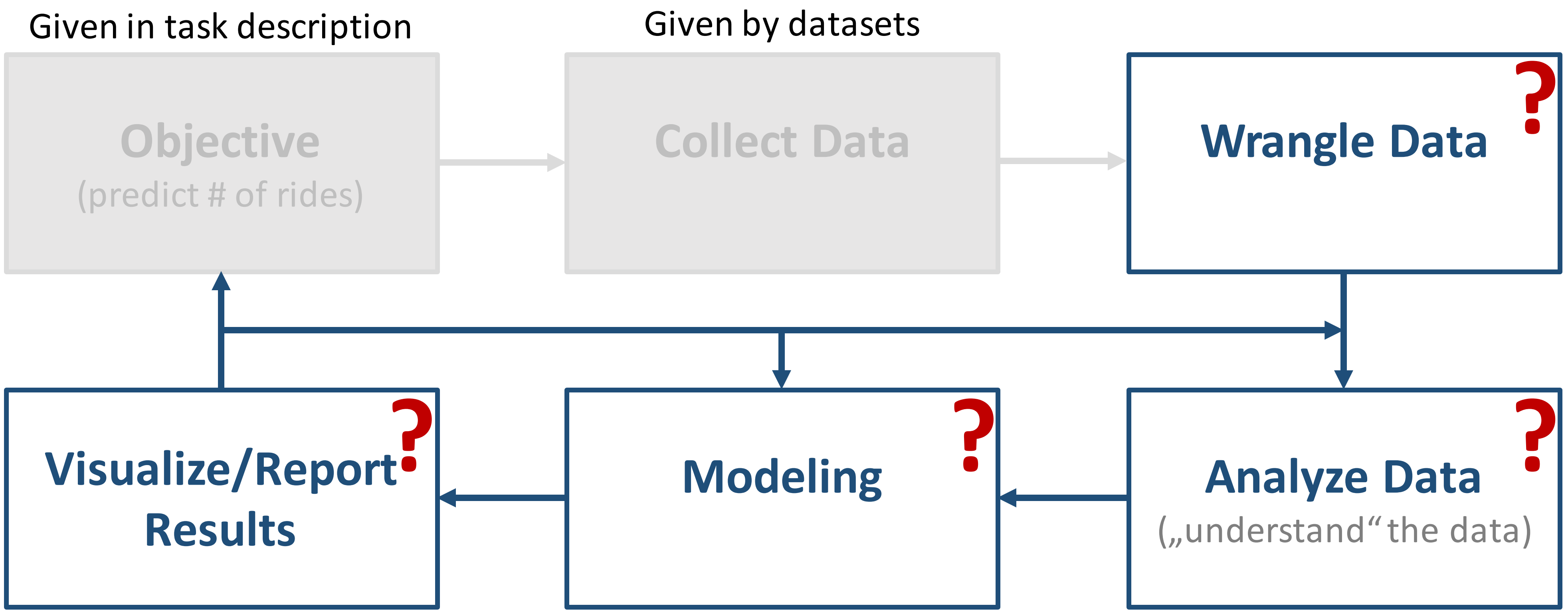
Practical Data Science: The Python Stack

Dr. David Koll

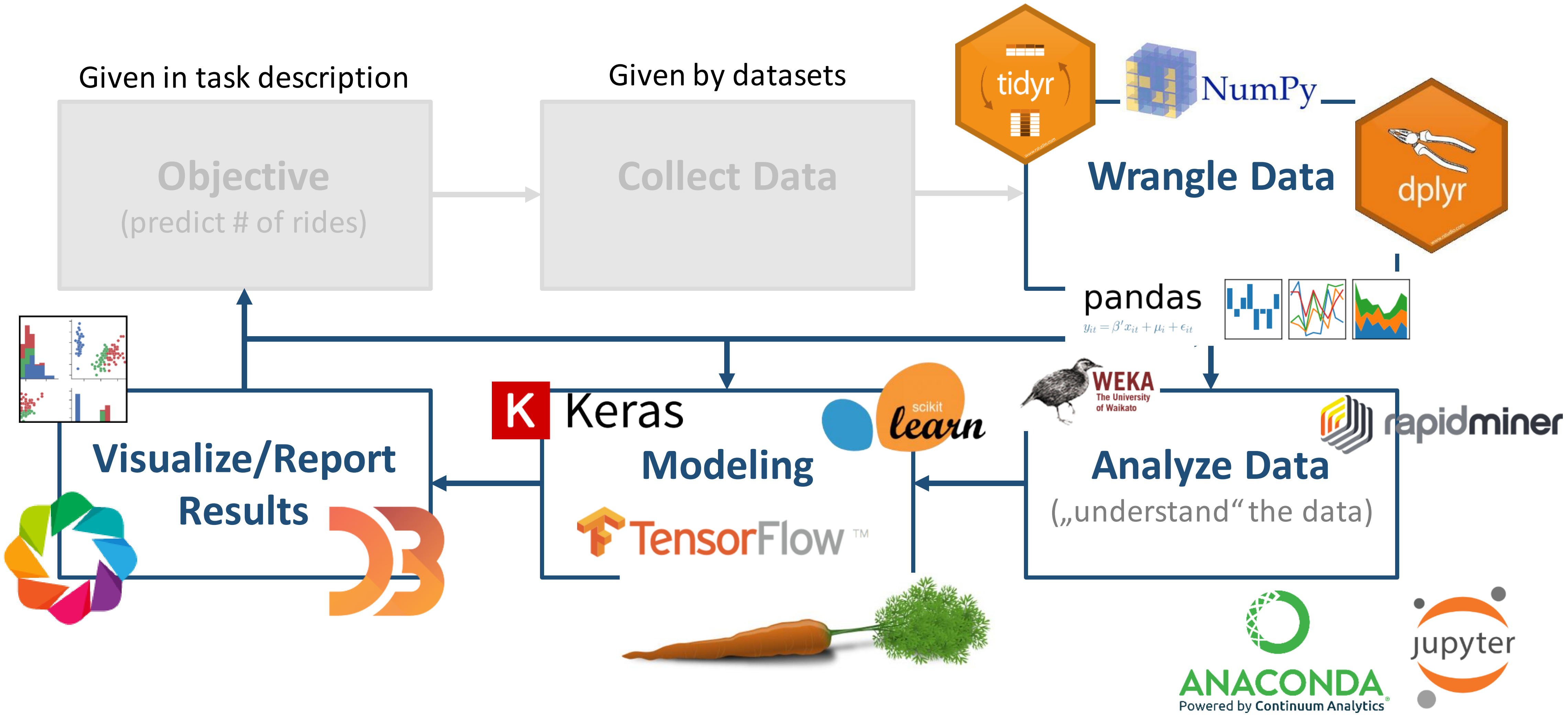
Organizational Stuff

- Room change is permanent: we will stay in this room (2.101)!
- Module: can be accredited for in Data Science study specialization (via special accreditation)
  - In subsequent semesters, we will have a new module specially designed for this

Recap: The DS Pipeline



Tools

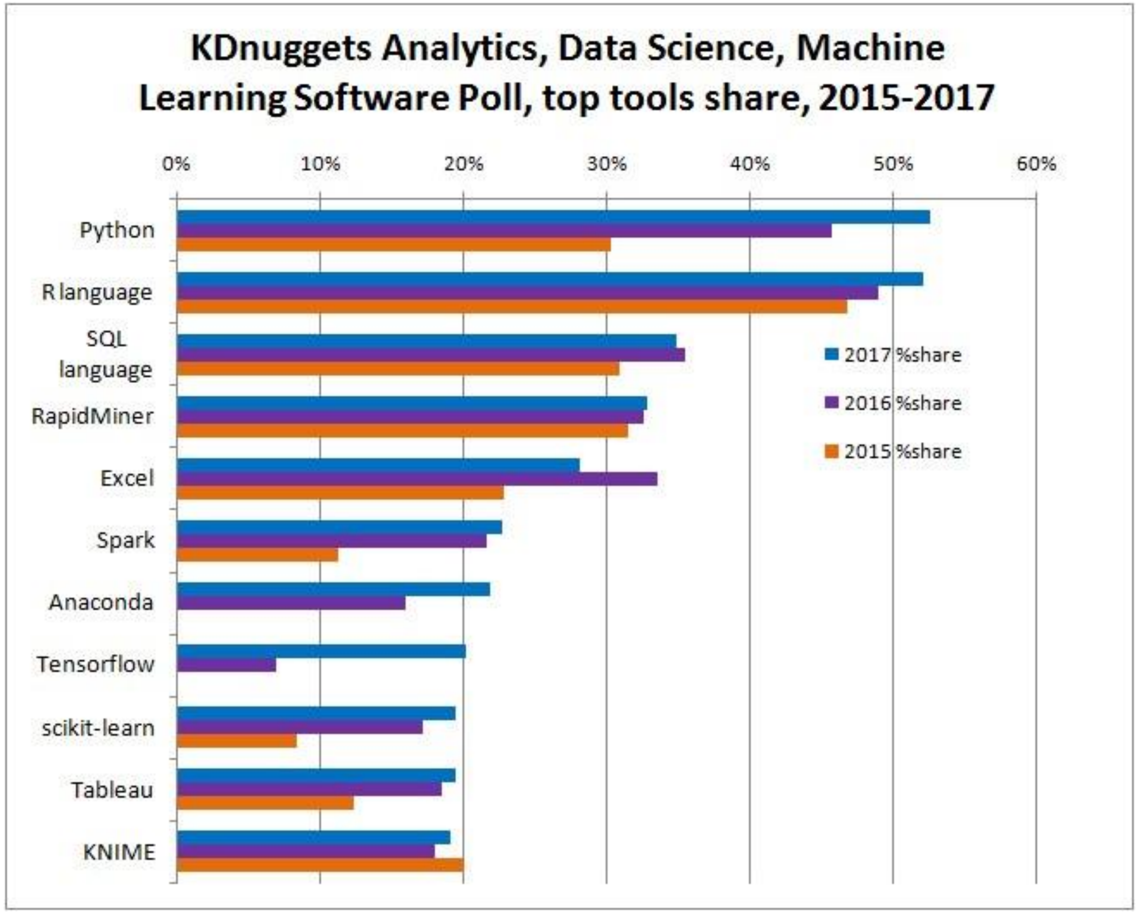

Given in task description
Objective
(predict # of rides)
Given by datasets
Collect Data
tidyr
NumPy
Wrangle Data
dplyr
pandas
Visualize/Report Results
K Keras
Modeling
TensorFlow™
scikit learn
WEKA
The University of Waikato
rapidminer
Analyze Data
(„understand“ the data)
ANACONDA
Powered by Continuum Analytics
jupyter

General Advice

- For beginners:
  - Getting to know one stack in detail is preferable over superficial ‘distributed’ knowledge
  - You can likely do anything you want to do (now) in every major stack
  - Differences are relatively minor in the beginning
  - Some languages are easier to learn than others

- For advanced data scientists (or for competitions):
  - Some algorithmic libraries are slightly more powerful than others
  - Some visualization libraries allow more expressiveness than others
  - Some tools are more convenient to use than others

First step: Picking a Language



http://www.kdnuggets.com/2017/05/poll-analytics-data-science-machine-learning-software-leaders.html

First step: Picking a Language

Table 1: Top Analytics/Data Science Tools in 2017 KDnuggets Poll

Tool	2017 % Usage	% change 2017 vs 2016	% alone
Python	52.6%	15%	0.2%
R language	52.1%	6.4%	3.3%
SQL language	34.9%	-1.8%	0%
RapidMiner	32.8%	0.7%	13.6%
Excel	28.1%	-16%	0.1%
Spark	22.7%	5.3%	0.2%
Anaconda	21.8%	37%	0.8%
Tensorflow	20.2%	195%	0%
scikit-learn	19.5%	13%	0%
Tableau	19.4%	5.0%	0.4%
KNIME	19.1%	6.3%	2.4%

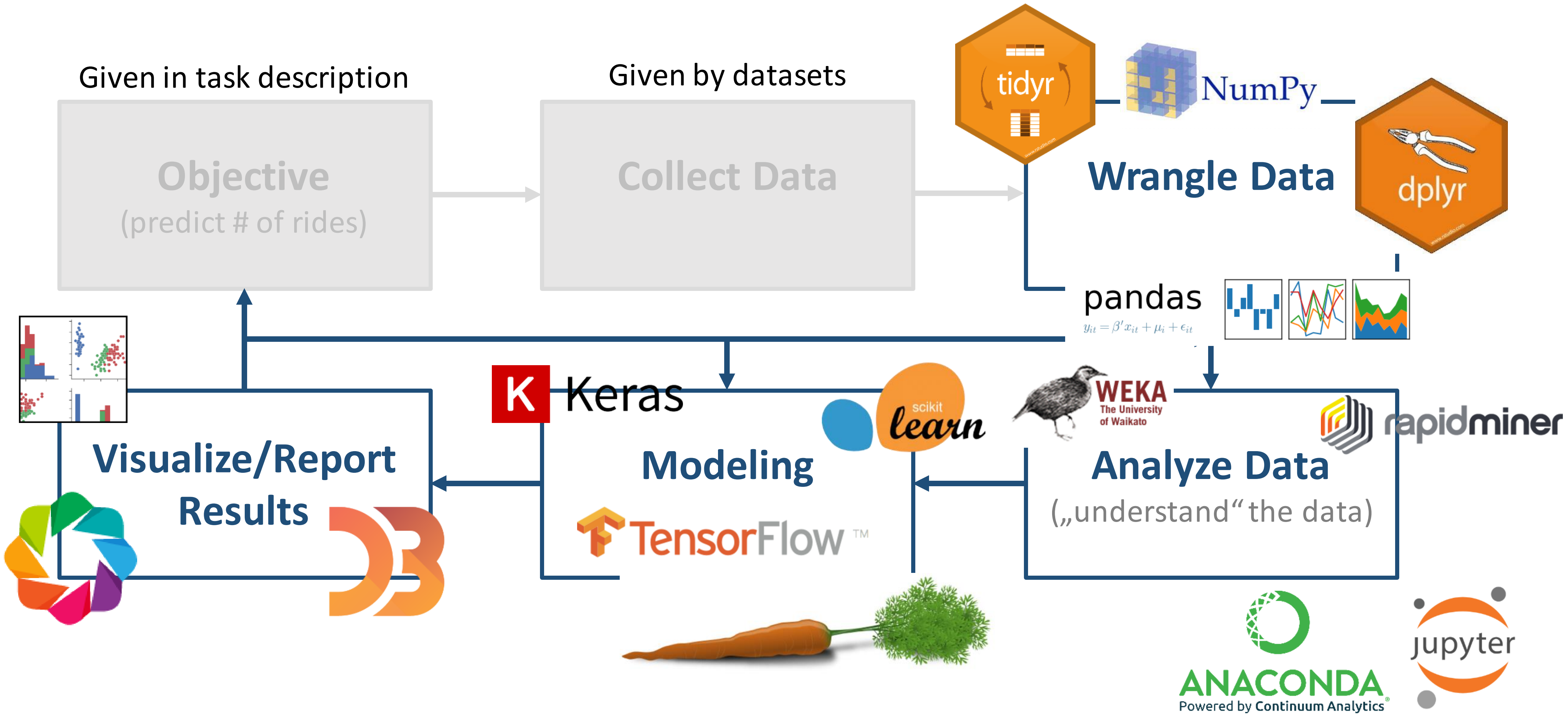
http://www.kdnuggets.com/2017/05/poll-analytics-data-science-machine-learning-software-leaders.html

Why Python in this Course?

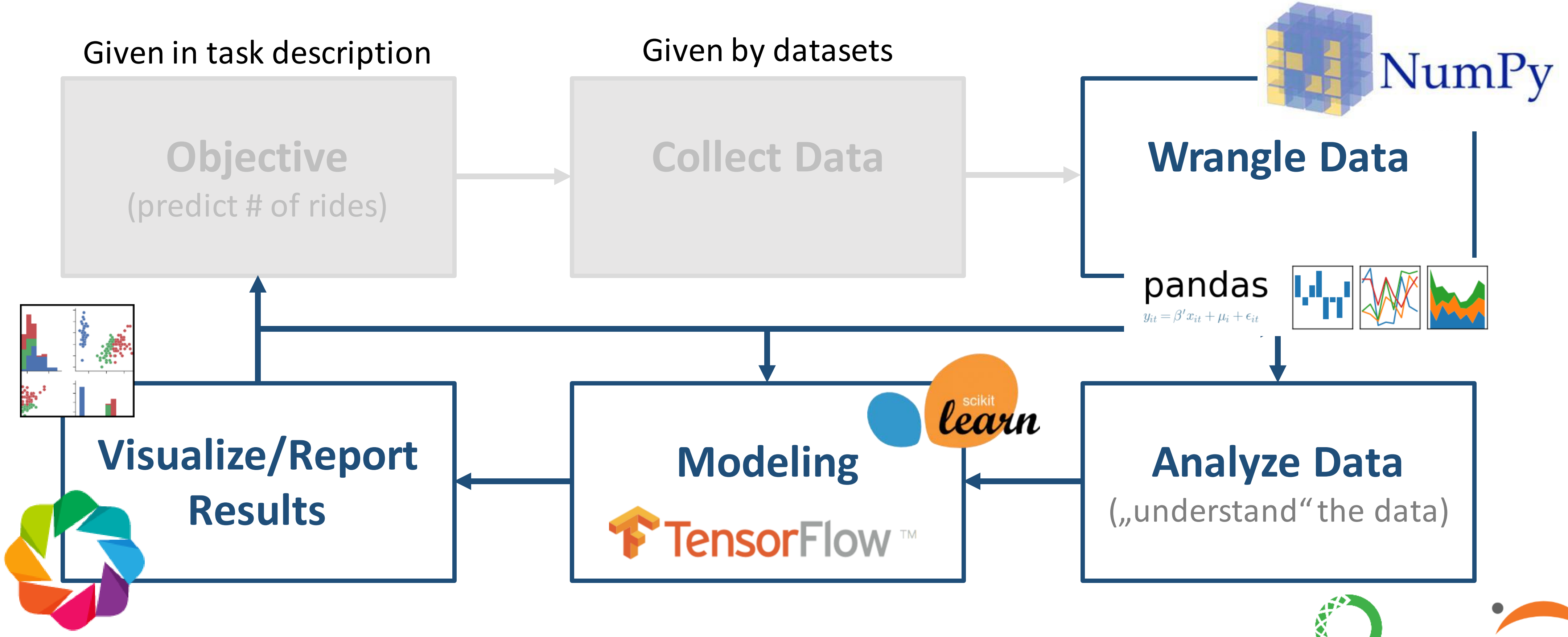
- Number one programming language in Data Science (according to KDN)
- Easier to learn than R, especially with CS background
- Anaconda offers easy package handling

- If you want to use something else, please feel free to do so

The Python Stack



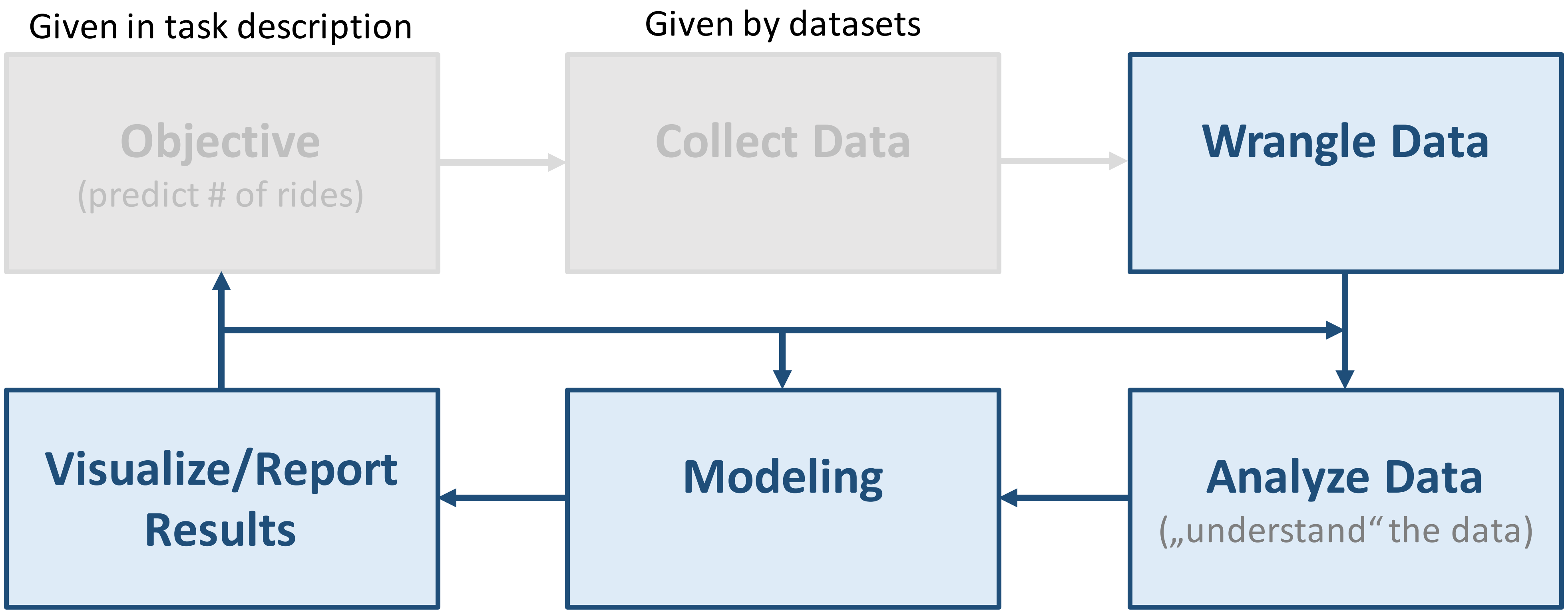
The Python Stack



ANACONDA
Powered by Continuum Analytics
jupyter

More Python libraries and tools: http://www.pydata.org

The Python Stack



ANACONDA
Powered by Continuum Analytics

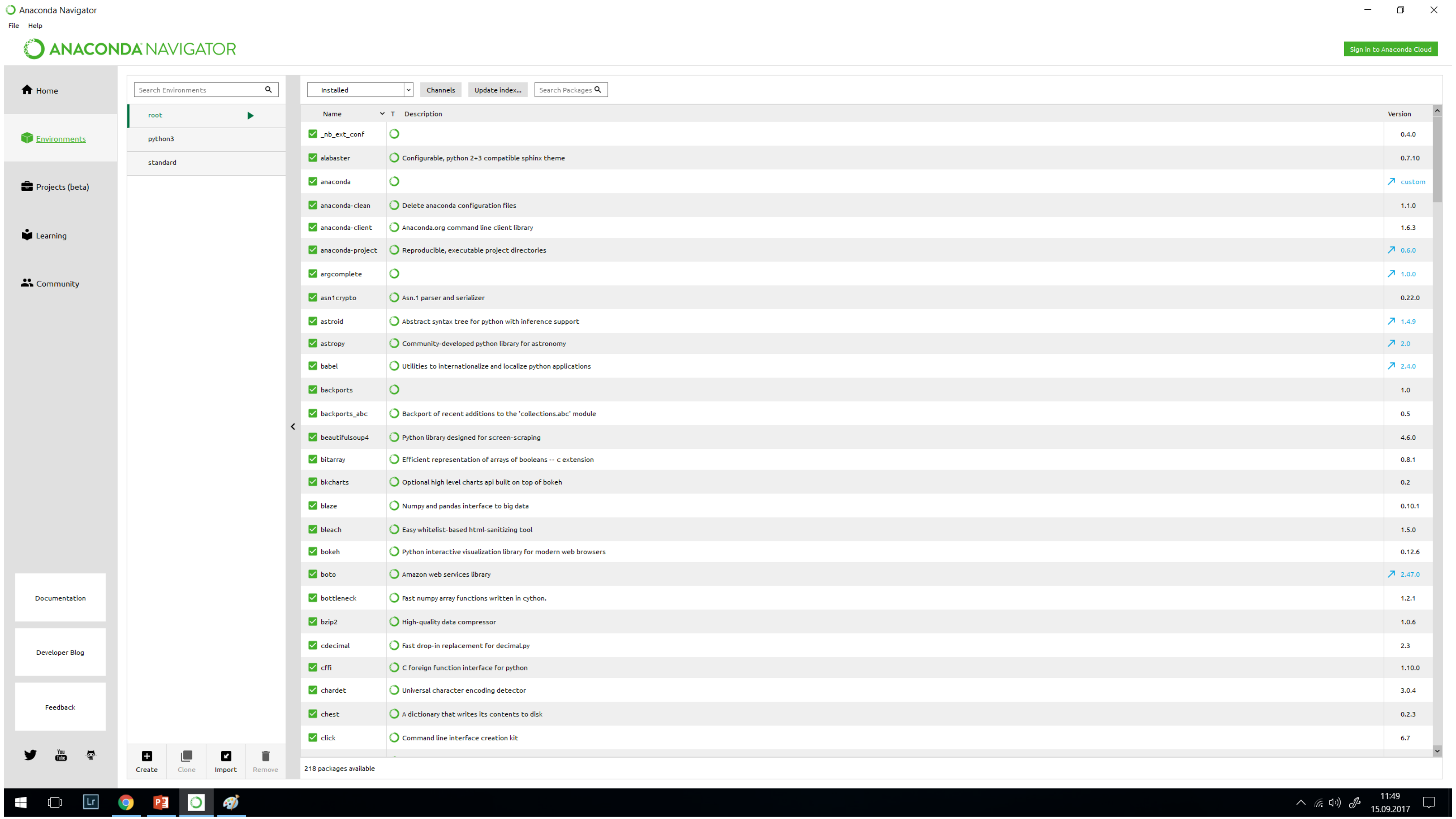
DS Environment Manager: Anaconda

- **Anaconda** is a Python distribution, featuring:
  - Package manager
    - 720 open source packages, largely related to data science
  - (Virtual) Environment manager

ANACONDA
Powered by Continuum Analytics®

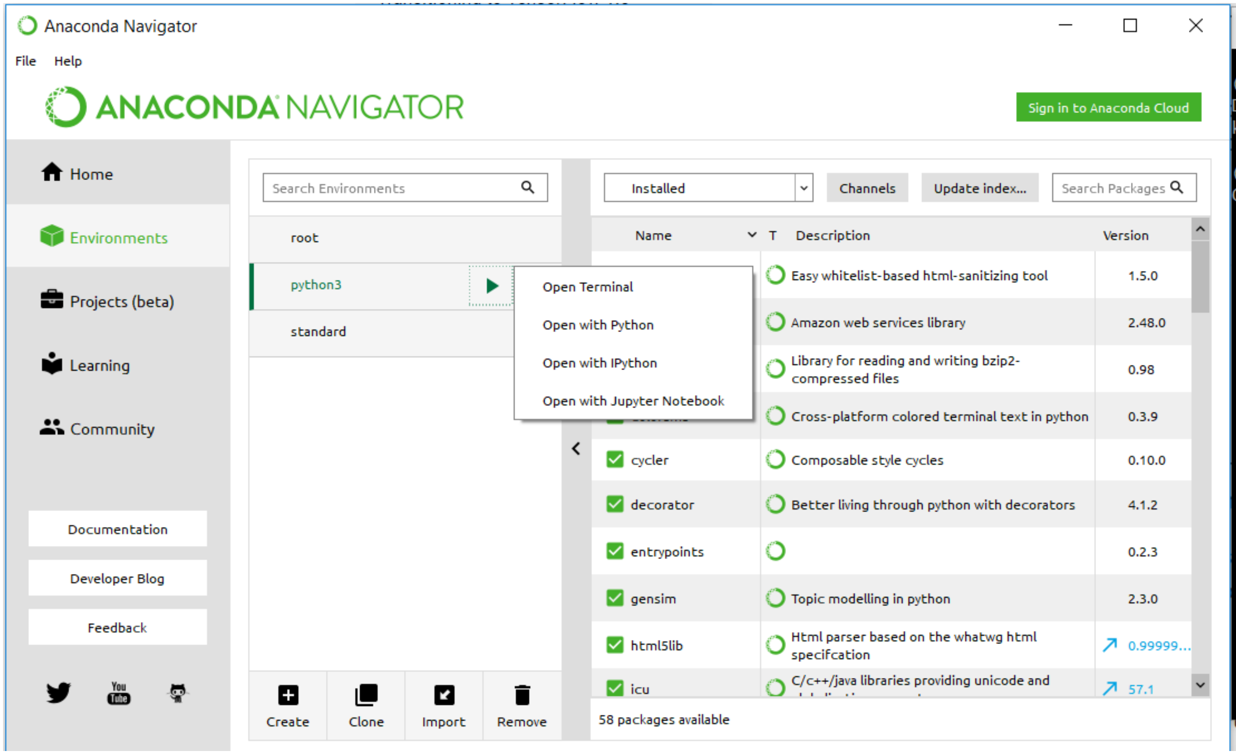
- Makes DS projects easier to handle:
  - Can have a different environment (e.g., Python 2 vs Python 3) for different projects
    - Also allows for different versions of a particular package
  - Each (virtual) environment can have different packages installed
    - Some DS packages do have interfering dependencies
  - Can import pre-defined images (e.g., Kaggle image for competitive data science)

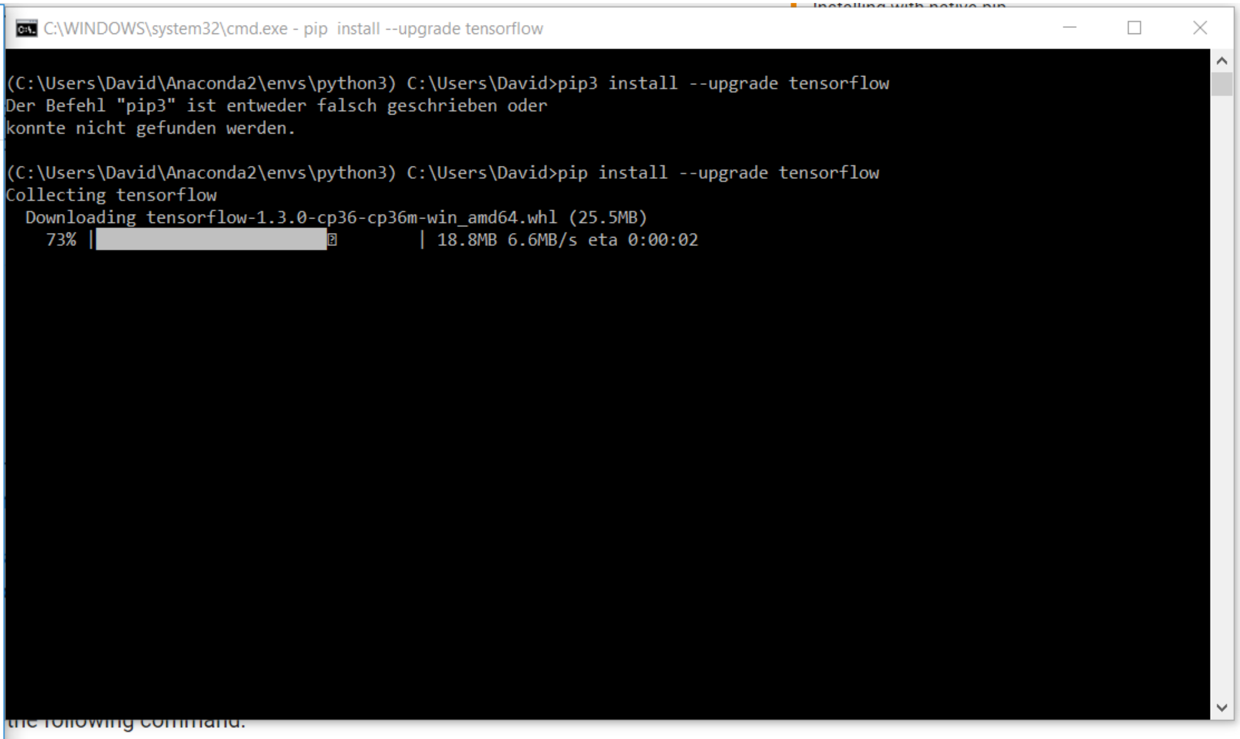
Anaconda: Navigator



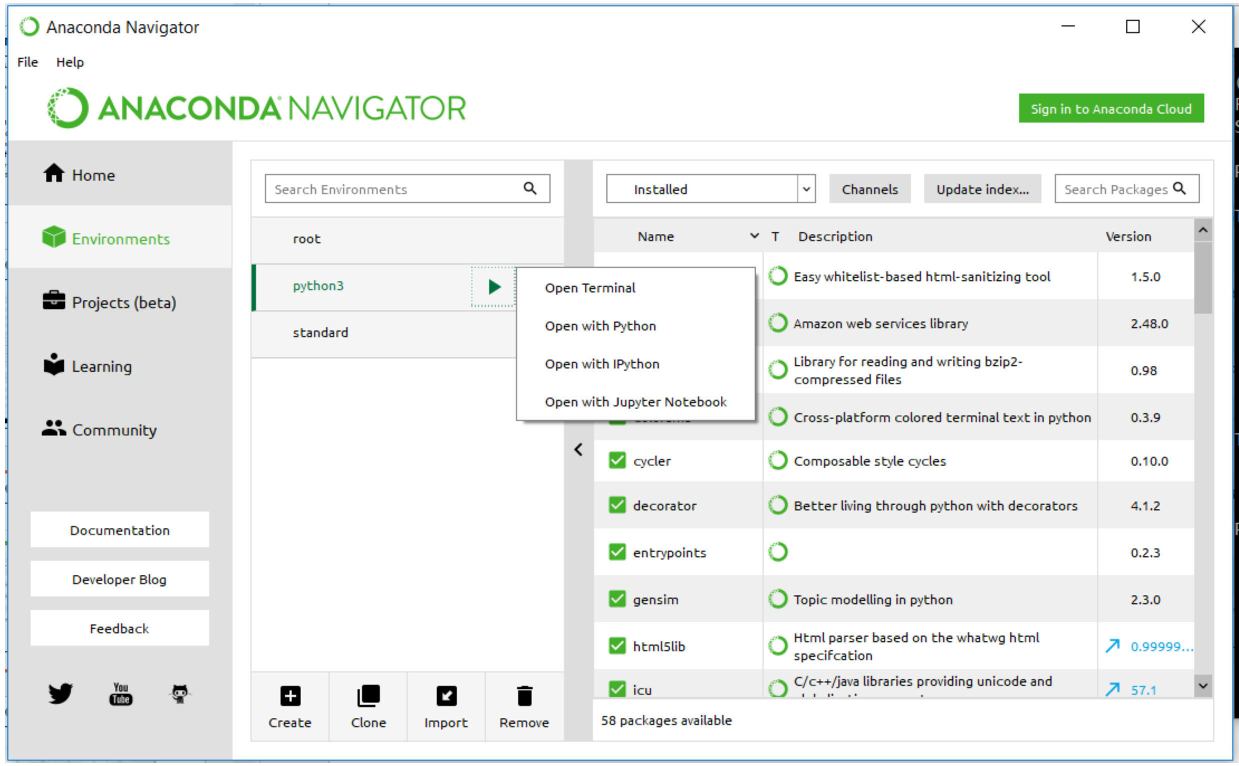
GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

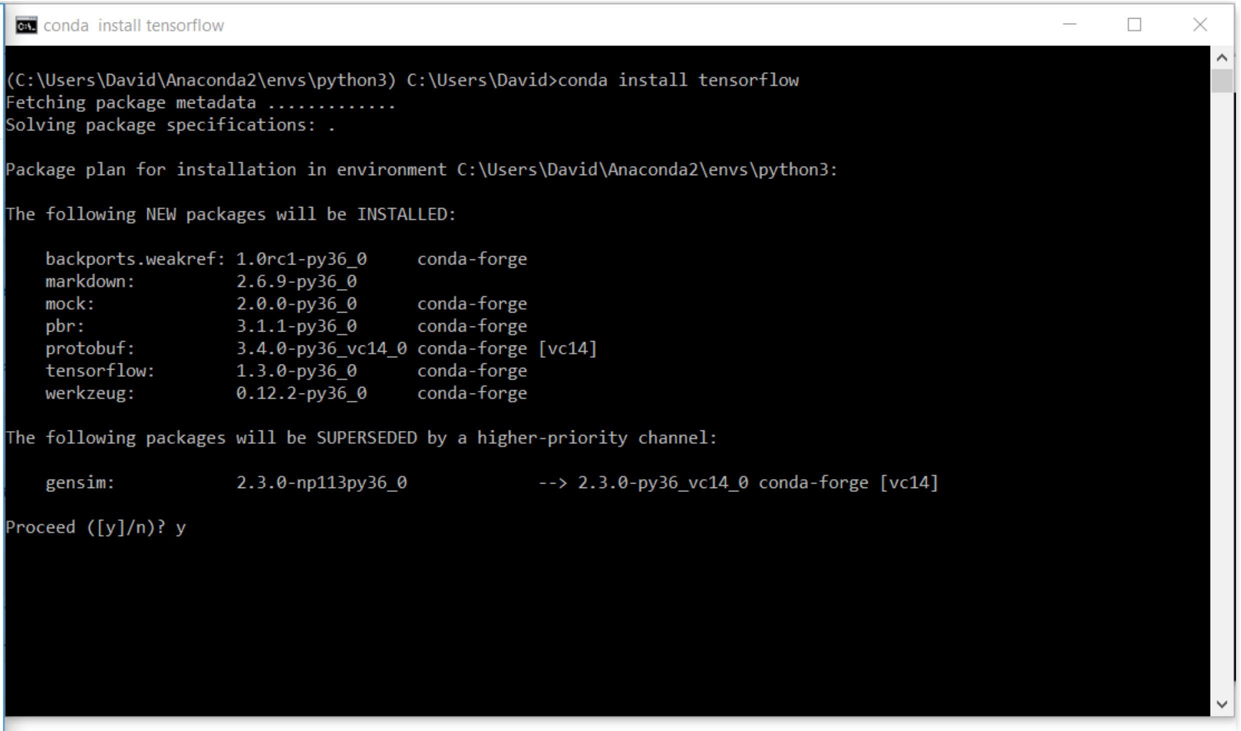
Anaconda: Navigator





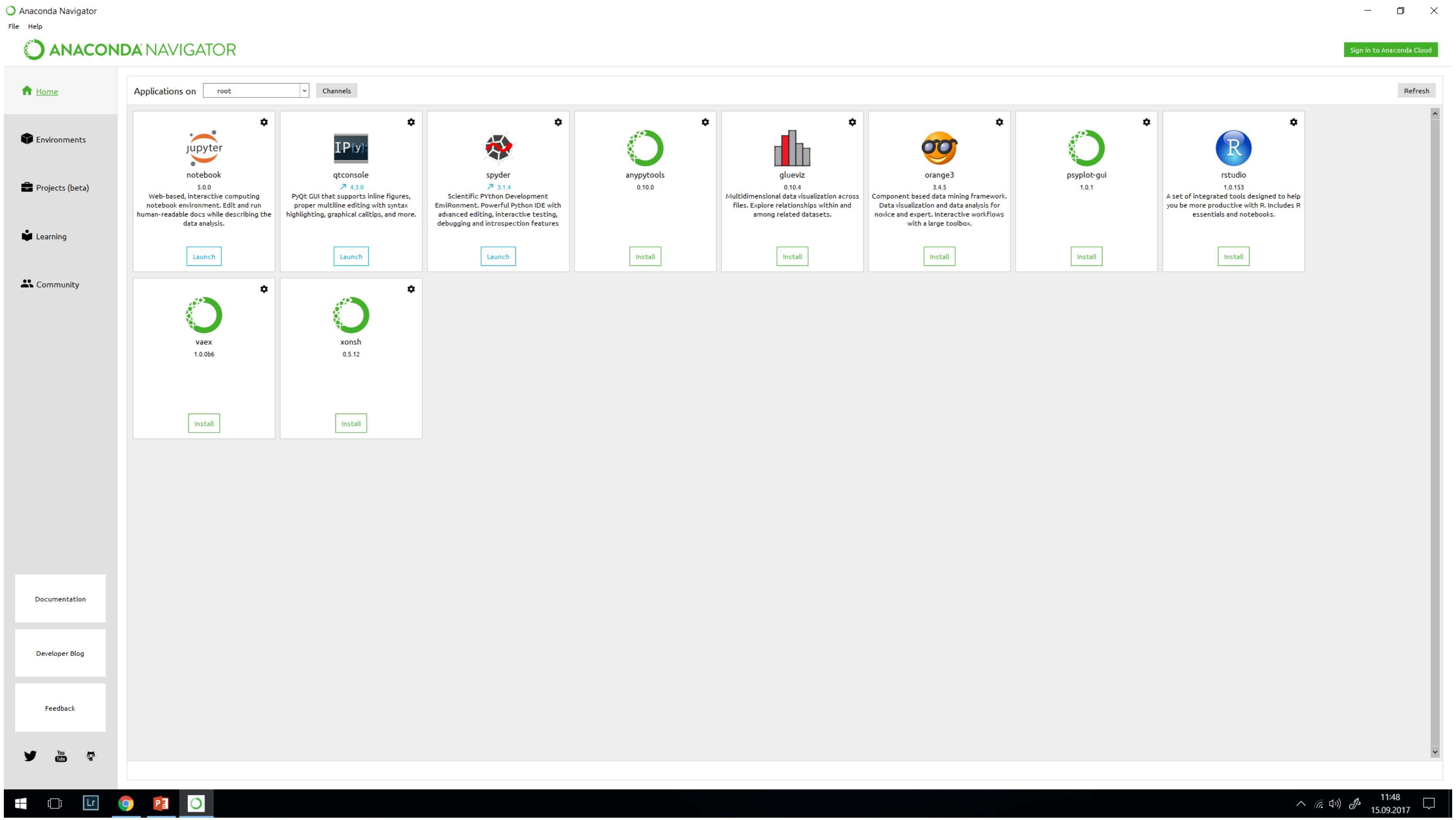
Anaconda: Navigator



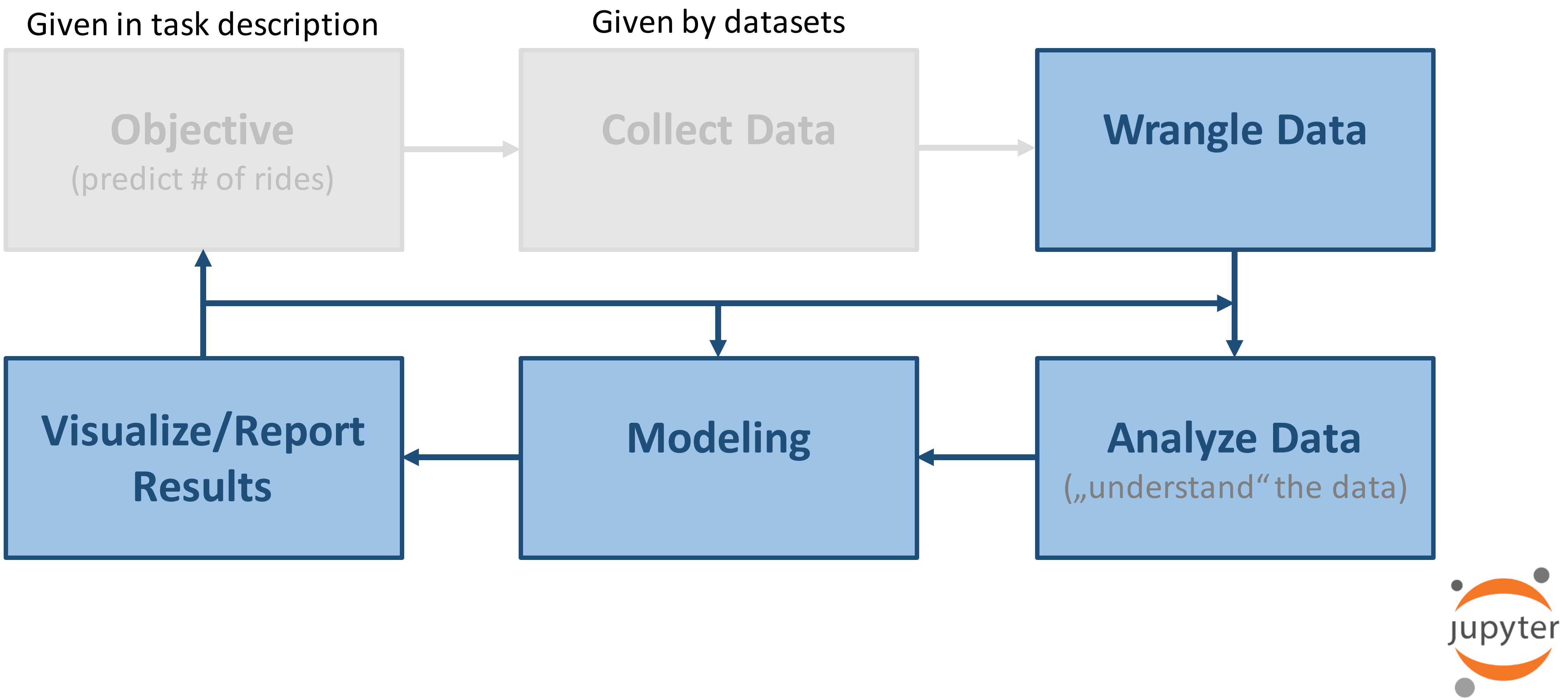


- Contrary to pip, conda installs from binaries (easier especially on Windows)
- General advice: try to conda install first, only run pip if conda fails

Anaconda: Navigator



The Python Stack



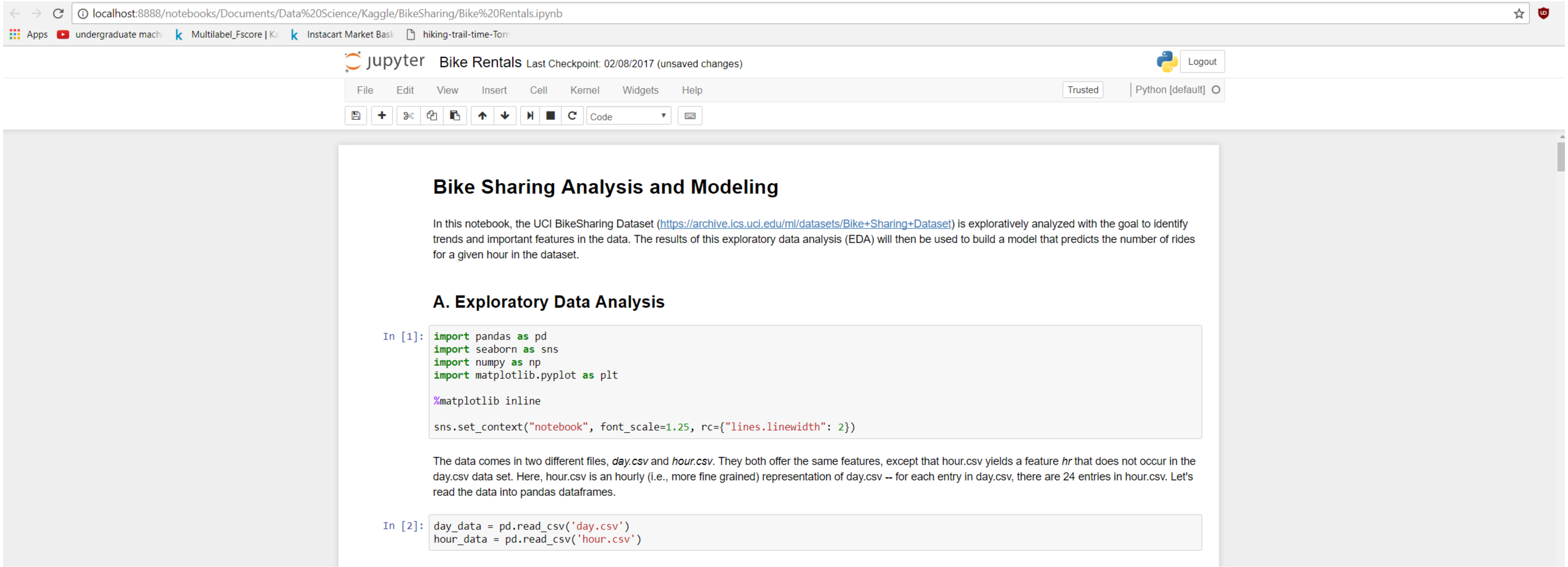
Jupyter Notebook

- **Jupyer Notebook** is a web-service tool well suited for DS:

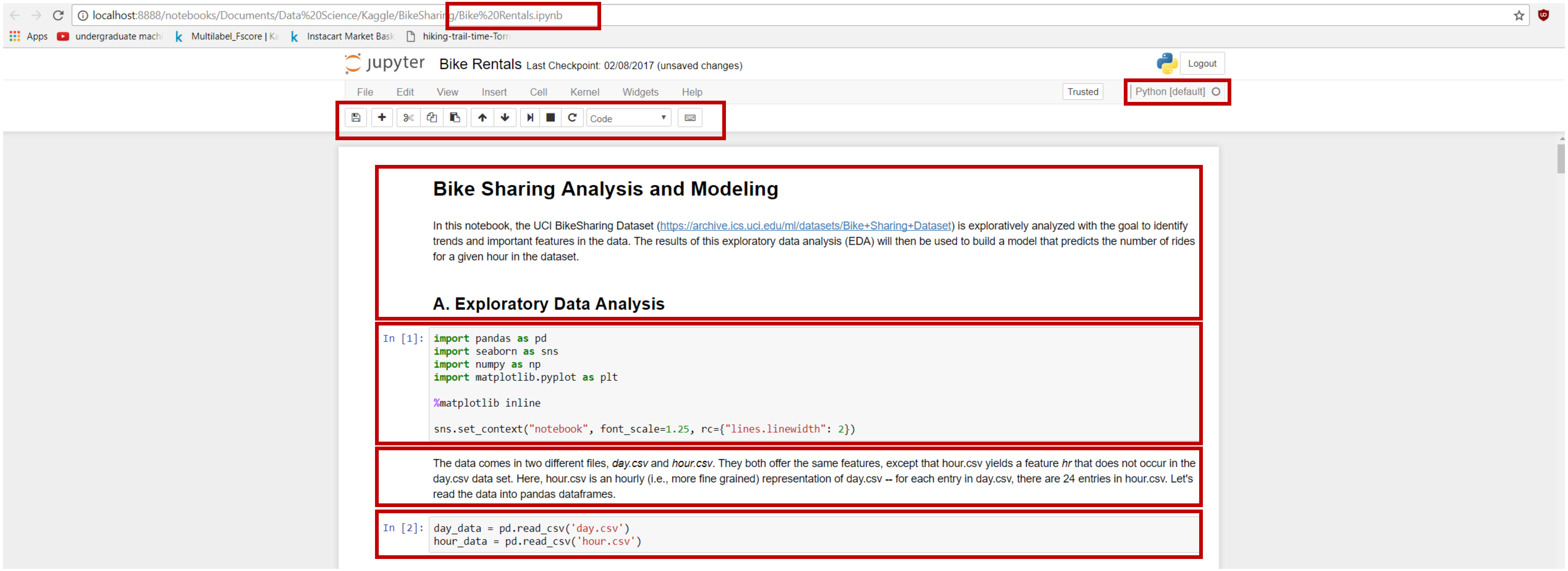
  "The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more."
  - http://jupyter.org/

- All the projects in this course can be done exclusively in Jupyter Notebooks
- Similar tool in R available since 12/2016: R Notebooks
  - However: Jupyter allows 40 different languages (including R), Spark integration etc.
    - See: https://try.jupyter.org/

Jupyter Notebook



Jupyter Notebook



Jupyter Notebook

jupyter Bike Rentals Last Checkpoint: 02/08/2017 (unsaved changes)

Logout

File Edit View Insert Cell Kernel Widgets Help

Python [default]

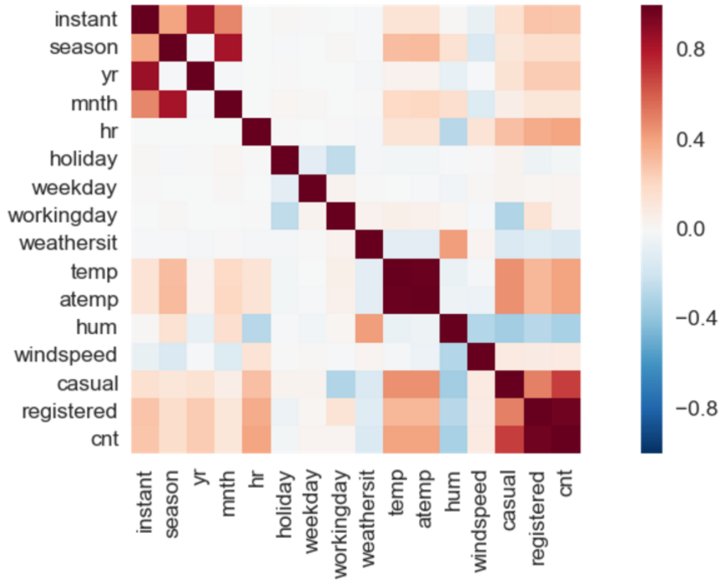
Markdown

2) Feature correlation

Before going into detail with the investigation of trends, one helpful visualization is to have a look at the correlation between features. This step is also important for building a model in the next step. For instance, when using linear regression, there should be no collinearity in the independent variables.

In [8]:

```
# get correlation matrix and visualize it in figure
cmat = hour_data.corr()

f, ax = plt.subplots(figsize=(10, 5))
sns.heatmap(cmat, square=True)
f.tight_layout()
```



Jupyter Notebook

jupyter Bike Rentals Last Checkpoint: 02/08/2017 (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Python [default]

Markdown

```
train = train.drop(['registered', 'casual', 'cnt', 'dteday'], axis=1)
test = test.drop(['registered', 'casual', 'cnt', 'dteday'], axis=1)
```

i) Model A...

... will predict the number of registered users for a specific hour. In this task we will use a single model (Random Forest Regression) to do this task. More advanced approaches could use meta models that take as an input the predictions of several single models and then apply some weighted averaging function on these predictions to come up with a final prediction.

For a first try, we will use RandomForest as, for instance, they do not assume linearity in the features, they can handle categorical features, and they do not easily overfit and thus typically yield much better generalization results even with a high number of regression trees employed.

In [26]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error
```

As a metric, we will use Mean Absolute Error (MAE). We could directly use the MAE criterion for the regressor, however, this currently seems to have severe drawbacks and will not be fixed before sklearn 0.19 (see, e.g.: https://www.kaggle.com/c/allstate-claims-severity/forums/t/24293/sklearn-randomforestregressor-mae-criterion). Therefore, we need to apply the MSE criterion when training our model. Note that this incurs high penalties on extreme cases (outliers), which in this case is not desired.

In [27]:
```
rfr = RandomForestRegressor(n_estimators=100, criterion='mse')
rfr.fit(train,target_train_registered)
```

Out[27]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_split=1e-07, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=100, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

In [28]:
```
print 'MAE: ', mean_absolute_error(target_test_registered, rfr.predict(test))
```

```
MAE:  19.1431933257
```

Textual Information: Markdown

- **Markdown** is a very simple text formatting syntax
  - Can easily build headings, tables, lists, formulas

```
### 6) Casual vs. Registered Users

Finally, we will provide some further insights (apart from the time of day) into how casual and registered users are using the
service. For that, we introduce a metric that measures the fraction of casual users participating in the service at a given time
as

$$\text{cas_frac}  = \frac{\text{# casual users}}{\text{# total users}}$$

In winter months (blue shaded areas in the plot), registered users are using the service almost exclusively, with only rare
occurances of higher casual user counts (*cas_frac* usually < 0.1); in warmer months, higher fractions of casual users appear.
Additionally, casual users are much more frequent on holidays and weekends, while days with high fractions of registered users
are usually working days. There are some interesting outliers that may be worth investigating.
```

6) Casual vs. Registered Users

Finally, we will provide some further insights (apart from the time of day) into how casual and registered users are using the service. For that, we introduce a metric that measures the fraction of casual users participating in the service at a given time as

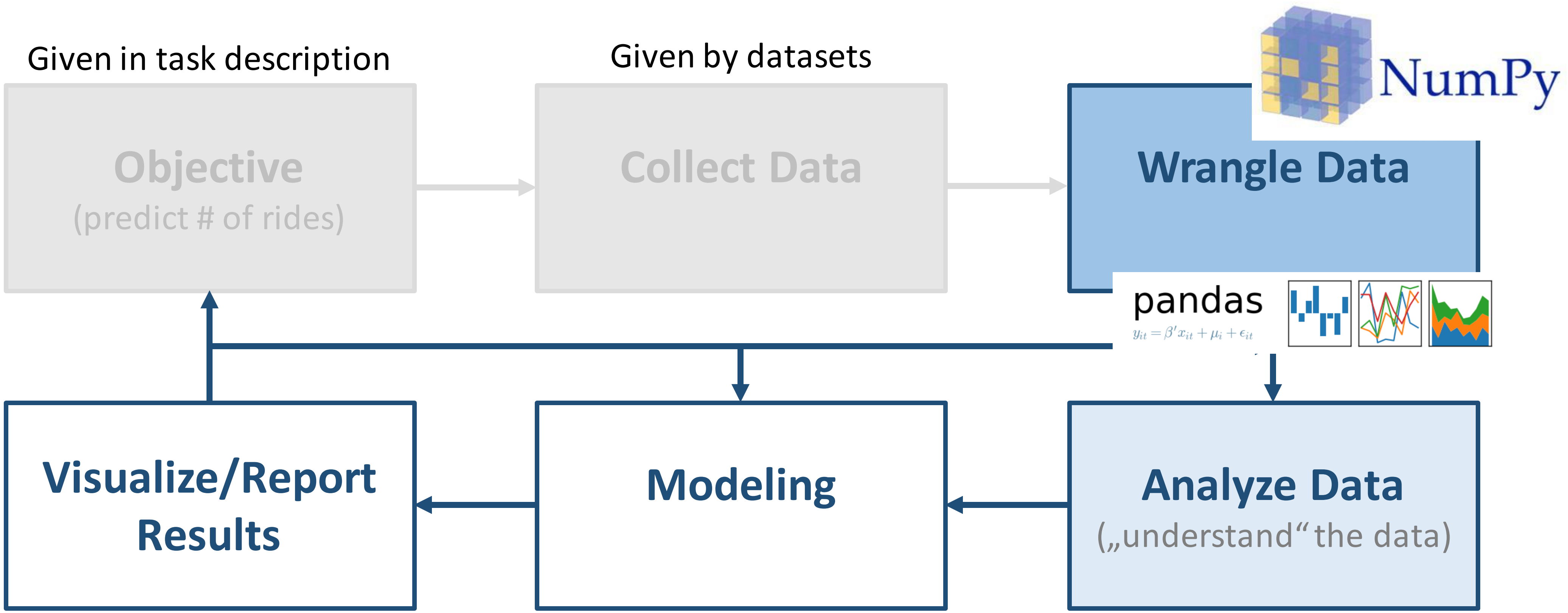
$$\text{cas\_frac} = \frac{\text{\# casual users}}{\text{\# total users}}$$

In winter months (blue shaded areas in the plot), registered users are using the service almost exclusively, with only rare occurances of higher casual user counts (*cas_frac* usually < 0.1); in warmer months, higher fractions of casual users appear. Additionally, casual users are much more frequent on holidays and weekends, while days with high fractions of registered users are usually working days. There are some interesting outliers that may be worth investigating.

Textual Information: Markdown

- Markdown cheat sheets:
  - https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
  - https://github.com/cben/mathdown/wiki/math-in-markdown

The Python Stack



Data Wrangling & Analysis: Pandas

- **Pandas** is the go-to library for handling (and analyzing) your data
  - Built on top of NumPy, a fundamental scientific computing package for python

- Offers functionality to…
  - read data
  - organize data in table-like data structures (dataframes)
  - manipulate data(frames)
  - aggregate statistics about data
  - and more

- …in a very fast and efficient way

NumPy in a Nutshell

- **NumPy** offers *ndarray:*
  - Object to encapsulate n-dimensional arrays

- Difference to standard python datastructures
  - Needs homogeneous data
  - Fixed size at initialization
  - More efficient handling of larger data due to using pre-compiled C code

- Most important for data wrangling: vectorization and broadcasting
  - Vectorization avoids any (slow) loops
  - Broadcasting allows element-by-element operations

NumPy Examples

- Vectorization*:*

```
c = []
for i in range(len(a)):
        c.append(a[i]  * b[i])
```

Standard Python

```
c = a*b
```

Numpy

- Broadcasting:

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a*b
```

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a*b
```

  - Both yield an equivalent result, b is *stretched* on the right
  - But: right side is 10% faster as it moves less memory

Pandas

- In most projects, pandas is the first library you will use
- Example: read in .csv data

```
import pandas as pd

df = pd.read_csv(`example_data.csv‘)
```

  - creates a pandas *DataFrame* (df), ****the**** data structure of pandas
  - df can then be manipulated further

- When reading data, pandas offers integrated handling of data alignment and missing data

Pandas Data Structures

- Besides *DataFrames*, pandas offers *Series:*
  - 1d-array, labeled (with an index)
  - Can hold any type of data
  - Similar to *ndarray* of NumPy, can call several ndarray functions on Series
    - Hence, can use optimized NumPy functions

- *DataFrames* can be seen as:
  - A Python dictionary of *Series* objects
  - More intuitively: SQL table

Pandas DataFrame

- Construction: Either from data file, or from Series/dict/…

```
d = {
   'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
   'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])
}

df = pd.DataFrame(d)
```

Output:

```
   one   two
a  1.0   1.0
b  2.0   2.0
c  3.0   3.0
d  NaN   4.0
```

Pandas DataFrame

- Quick view at the constructed frame
  - df.head(n) – returns first n rows of dataset (default = 5)
  - df.tail(n) – returns last n rows of dataset (default = 5)
  - df.describe() – returns basic statistical information

```
In [93]: hour_data.head()
```

Out[93]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0.0	3	13	16
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.22	0.2727	0.80	0.0	8	32	40
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.22	0.2727	0.80	0.0	5	27	32
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0.0	3	10	13
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0.0	0	1	1

Pandas DataFrame

- Quick view at the constructed frame
  - df.head(n) – returns first n rows of dataset (default = 5)
  - df.tail(n) – returns last n rows of dataset (default = 5)
  - df.describe() – returns basic statistical information

In [91]:
```
hour_data.describe()
```

Out[91]:

	instant	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp
count	17379.0000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.00(
mean	8690.0000	2.501640	0.502561	6.537775	11.546752	0.028770	3.003683	0.682721	1.425283	0.496987
std	5017.0295	1.106918	0.500008	3.438776	6.914405	0.167165	2.005771	0.465431	0.639357	0.192556
min	1.0000	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.020000
25%	4345.5000	2.000000	0.000000	4.000000	6.000000	0.000000	1.000000	0.000000	1.000000	0.340000
50%	8690.0000	3.000000	1.000000	7.000000	12.000000	0.000000	3.000000	1.000000	1.000000	0.500000
75%	13034.5000	3.000000	1.000000	10.000000	18.000000	0.000000	5.000000	1.000000	2.000000	0.660000
max	17379.0000	4.000000	1.000000	12.000000	23.000000	1.000000	6.000000	1.000000	4.000000	1.000000

Pandas DataFrame

- Indexing

```
df[`one`]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas DataFrame

- Indexing

```
df[`one`]
df[`two`]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas DataFrame

- Indexing

```
df[`one`]
df[`two`]

df[:2]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas DataFrame

- Indexing

```
df[`one`]
df[`two`]

df[:2]
df.iloc[3]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas DataFrame

- Indexing

```
df[`one`]
df[`two`]

df[:2]
df.iloc[3]

df[df[`one`] < 2]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas DataFrame

- Indexing

```
df[`one`]
df[`two`]

df[:2]
df.iloc[3]

df[df[`one`] < 2]
df[df[`one`] < 2][`two`]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas DataFrame

- Indexing

```
df[`one`]
df[`two`]

df[:2]
df.iloc[3]

df[df[`one`] < 2]
df[df[`one`] < 2][`two`]
```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

- Many other indexing operations possible
  - Check out the documentation:
    - https://pandas.pydata.org/pandas-docs/stable/dsintro.html

Pandas DataFrame

- Recall: pandas is based on NumPy
  - Element-wise operations
  - Very convenient for feature engineering

```
df[`ratio`] = df[`one`] / df[`two`]
```

	one	two	ratio
a	1.0	1.0	1.0
b	2.0	2.0	1.0
c	3.0	3.0	1.0
d	NaN	4.0	NaN

Pandas – SQL-like Data Queries

- Indexing as one example for SQL *where*
- In general: use filters to select subset of data
  - Syntax: df[<filter expression>]
  - <filter expression> can be many things, e.g.:
    - Range:
    - Boolean:
    - Combinations:

```
hour_data[hour_data.weekday < 6]
hour_data[hour_data.season == 1]
hour_data[(hour_data.holiday == 0) & (hour_data.hr == 7)]
```

In [93]:
```
hour_data.head()
```

Out[93]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
0	1	2011-01-01	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0.0	3	13	16
1	2	2011-01-01	1	0	1	1	0	6	0	1	0.22	0.2727	0.80	0.0	8	32	40
2	3	2011-01-01	1	0	1	2	0	6	0	1	0.22	0.2727	0.80	0.0	5	27	32
3	4	2011-01-01	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0.0	3	10	13
4	5	2011-01-01	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0.0	0	1	1

Pandas – SQL-like Data Queries

- SQL Insert column

```
df[`three`] = [`I`, `am`, `an`, `Insert`]
```

	one	two	three
a	1.0	1.0	I
b	2.0	2.0	am
c	3.0	3.0	an
d	NaN	4.0	Insert

- SQL Join

```
df_to_join = pd.dataFrame(
        `two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c‘, `d`]),
        `four' : pd.Series([4., 3., 2., 1.], index=['a', 'b', 'c', 'd'])
)
```

	two	four
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

Pandas – SQL-like Data Queries

- SQL Insert column

```
df[`three`] = [`I`, `am`, `an`, `Insert`]
```

	one	two	three
a	1.0	1.0	I
b	2.0	2.0	am
c	3.0	3.0	an
d	NaN	4.0	Insert

- SQL Join

```
df_to_join = pd.dataFrame(
        `two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c‘, `d`]),
        `four' : pd.Series([4., 3., 2., 1.], index=['a', 'b', 'c', 'd'])
)

df = pd.merge(df, df_to_join, on=`two`, how=`left`)
```

	one	two	three	four
a	1.0	1.0	I	4.0
b	2.0	2.0	am	3.0
c	3.0	3.0	an	2.0
d	NaN	4.0	Insert	1.0

Pandas – SQL-like Data Queries

- SQL Update -- as an example of apply()

```
df[`two`] = df[`two`].apply(lambda x: x**2)
```

```
def square(x):
      return(x**2)

df[`two`] = df[`two`].apply(square)
```

	one	two	three	four
a	1.0	1.0	I	4.0
b	2.0	4.0	am	3.0
c	3.0	9.0	an	2.0
d	NaN	16.0	Insert	1.0

- General advice: apply is usually much more efficient than:

```
squared = []
for x in df[`two`]:
      squared.append(x**2)
df[`two] = squared
```

- Apply still uses loops internally, but more efficiently implemented

Pandas – Optimization One Step Further

- Vectorization (only works if all calls in function are vectorized)

```
df[`two`] = df[`two`].apply(lambda x: x**2)
```

```
def square(x):
    return(x**2)

df[`two`] = square(df[`two`])
```

	one	two	three	four
a	1.0	1.0	I	4.0
b	2.0	4.0	am	3.0
c	3.0	9.0	an	2.0
d	NaN	16.0	Insert	1.0

- Vectorization + numpy arrays (removes pandas indexing overhead)

```
def square(x):
    return(x**2)

df[`two`] = square(df[`two`].values)
```

- Potential gain from looping to vectorized numpy: >1000x

Pandas – SQL-like Data Queries

- SQL Groupby
  - Note groupby returns a DataFrameGroupBy object
  - You need to call some aggregation function on that object
  - Index will be created based on groupby key
    - In most cases, you want to reset the index to get a nice format (e.g., join-able with other df)

```
In [5]: hour_data.groupby('season').mean().reset_index()
Out[5]:
```

	season	instant	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual
0	1	6302.008015	0.512494	3.119755	11.648515	0.038661	3.008722	0.658652	1.460160	0.299147	0.298116	0.581348	0.215107	14.290901
1	2	7287.727376	0.500340	4.654117	11.512134	0.021774	2.991608	0.695396	1.443638	0.544663	0.520547	0.627022	0.203410	46.160581
2	3	9526.588968	0.501779	7.689724	11.507562	0.021352	3.033141	0.698621	1.330294	0.706410	0.656004	0.633167	0.171593	50.287144
3	4	11655.779301	0.495747	10.702505	11.522448	0.034026	2.979915	0.676749	1.472117	0.423138	0.415738	0.667124	0.170819	30.666824

- Alternative aggregations: median, count, min, max, nunique, sum, …

Pandas – SQL-like Data Queries

- SQL Groupby
  - Can also group by two or more columns

In [14]:
```
hour_data.groupby(['yr','season']).mean()
```

Out[14]:

yr	season	instant	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered
0	1	1864.575919	3.156673	11.764507	0.034333	3.080271	0.657640	1.457930	0.275348	0.276990	0.574623	0.215586	10.360251	62.173598
	2	2909.000000	4.655470	11.517022	0.021788	2.982297	0.695415	1.480254	0.534607	0.510330	0.658311	0.205680	35.208352	122.447571
	3	5130.500000	7.687946	11.515179	0.021429	3.002232	0.705804	1.340179	0.701339	0.654150	0.644125	0.176337	42.611607	144.732143
	4	7317.500000	10.696813	11.508435	0.033739	2.989691	0.673852	1.476101	0.426354	0.418061	0.694016	0.168109	24.748360	128.080600
1	1	10523.079577	3.084637	11.538178	0.042778	2.940662	0.659614	1.462282	0.321785	0.318213	0.587746	0.214652	18.029899	129.784269
	2	11660.500000	4.652765	11.507253	0.021759	3.000907	0.695376	1.407072	0.554705	0.530750	0.595775	0.201144	57.097915	201.865367
	3	13891.500000	7.691489	11.500000	0.021277	3.063830	0.691489	1.320479	0.711445	0.657844	0.622287	0.166882	57.908245	226.435284
	4	16068.500000	10.708294	11.536702	0.034318	2.969971	0.679695	1.468065	0.419867	0.413375	0.639771	0.173575	36.686845	209.011916

Pandas – SQL-like Data Queries

- Combinations of filters and SQL-like statements

In [21]:
```
hour_data[(hour_data['hr'] > 2) & (hour_data['hr'] < 6) ].groupby(['yr','season']).mean()
```

Out[21]:

yr	season	instant	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered
0	1	1966.870968	3.313364	4.041475	0.036866	3.124424	0.631336	1.447005	0.237143	0.244939	0.653134	0.181310	0.599078	3.870968
	2	2916.339483	4.678967	4.000000	0.022140	3.011070	0.693727	1.494465	0.465314	0.449345	0.791255	0.165562	2.011070	8.029520
	3	5118.370504	7.683453	4.003597	0.021583	3.039568	0.708633	1.309353	0.636259	0.599371	0.765755	0.131543	2.744604	10.956835
	4	7311.661654	10.699248	4.003759	0.033835	2.988722	0.672932	1.447368	0.378120	0.376391	0.788684	0.138990	1.409774	9.548872
1	1	10496.844697	3.060606	4.022727	0.034091	2.977273	0.659091	1.496212	0.280227	0.283690	0.669318	0.179225	0.659091	8.651515
	2	11657.445255	4.656934	4.003650	0.021898	3.007299	0.693431	1.423358	0.490146	0.473564	0.713686	0.162278	2.021898	12.536496
	3	13884.000000	7.691489	4.000000	0.021277	3.063830	0.691489	1.304965	0.641277	0.598176	0.767199	0.127771	2.833333	17.056738
	4	16060.131274	10.710425	4.007722	0.034749	2.969112	0.675676	1.490347	0.369421	0.368487	0.738764	0.139117	1.694981	15.262548

Pandas: Data Wrangling

- Methods available for handling missing data:

fillna(value)

```
In [11]: df
Out[11]:
```

	0	1	2
0	1	2.0	3
1	20	21.0	10
2	1	NaN	3
3	2	3.0	4

```
In [13]: df[1] = df[1].fillna(df[1].mean())
In [14]: df
Out[14]:
```

	0	1	2
0	1	2.000000	3
1	20	21.000000	10
2	1	8.666667	3
3	2	3.000000	4

dropna()

```
In [16]: df
Out[16]:
```

	0	1	2
0	1	2.0	3
1	20	21.0	10
2	1	NaN	3
3	2	3.0	4

```
In [17]: df = df.dropna()
In [18]: df
Out[18]:
```

	0	1	2
0	1	2.0	3
1	20	21.0	10
3	2	3.0	4

interpolate()

```
In [20]: df
Out[20]:
```

	0	1	2
0	1	2.0	3
1	20	21.0	10
2	1	NaN	3
3	2	3.0	4

```
In [21]: df = df.interpolate(method='nearest')
In [22]: df
Out[22]:
```

	0	1	2
0	1	2.0	3
1	20	21.0	10
2	1	21.0	3
3	2	3.0	4

Pandas: Data Wrangling

- Methods for inconsistent data?
  - Mainly: df.describe()
  - Find out about inconsistent data in a different way?

- Inconsistent data, some preprocessing tasks: better use visualization and/or SciKit Learn

Pandas: Feature engineering

- Close to all feature engineering you do, you will do with pandas
  - Exception: learned features
- Differences, ratios, etc: one liner with pandas
- Similarly: dummy encoding for categorical variables
  - Some learners can not handle categoricals

In [3]:
```
hour_data.describe()
```

Out[3]:

	instant	season	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp
count	17379.0000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000	17379.000000
mean	8690.0000	2.501640	0.502561	6.537775	11.546752	0.028770	3.003683	0.682721	1.425283	0.496987
std	5017.0295	1.106918	0.500008	3.438776	6.914405	0.167165	2.005771	0.465431	0.639357	0.192556
min	1.0000	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.020000
25%	4345.5000	2.000000	0.000000	4.000000	6.000000	0.000000	1.000000	0.000000	1.000000	0.340000
50%	8690.0000	3.000000	1.000000	7.000000	12.000000	0.000000	3.000000	1.000000	1.000000	0.500000
75%	13034.5000	3.000000	1.000000	10.000000	18.000000	0.000000	5.000000	1.000000	2.000000	0.660000
max	17379.0000	4.000000	1.000000	12.000000	23.000000	1.000000	6.000000	1.000000	4.000000	1.000000

Pandas: Feature engineering

- Close to all feature engineering you do, you will do with pandas
  - Exception: learned features
- Differences, ratios, etc: one liner with pandas
- Similarly: dummy encoding for categorical variables
  - Some learners can not handle categoricals

In [15]:
```
season_data = pd.get_dummies(hour_data['season'])
hour_data.drop(['season'], axis=1, inplace=True)
hour_data = hour_data.join(season_data)
hour_data.head()
```

Out[15]:

	instant	dteday	yr	mnth	hr	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt	1	2	3	4
0	1	2011-01-01	0	1	0	0	6	0	1	0.24	0.2879	0.81	0.0	3	13	16	1	0	0	0
1	2	2011-01-01	0	1	1	0	6	0	1	0.22	0.2727	0.80	0.0	8	32	40	1	0	0	0
2	3	2011-01-01	0	1	2	0	6	0	1	0.22	0.2727	0.80	0.0	5	27	32	1	0	0	0
3	4	2011-01-01	0	1	3	0	6	0	1	0.24	0.2879	0.75	0.0	3	10	13	1	0	0	0
4	5	2011-01-01	0	1	4	0	6	0	1	0.24	0.2879	0.75	0.0	0	1	1	1	0	0	0

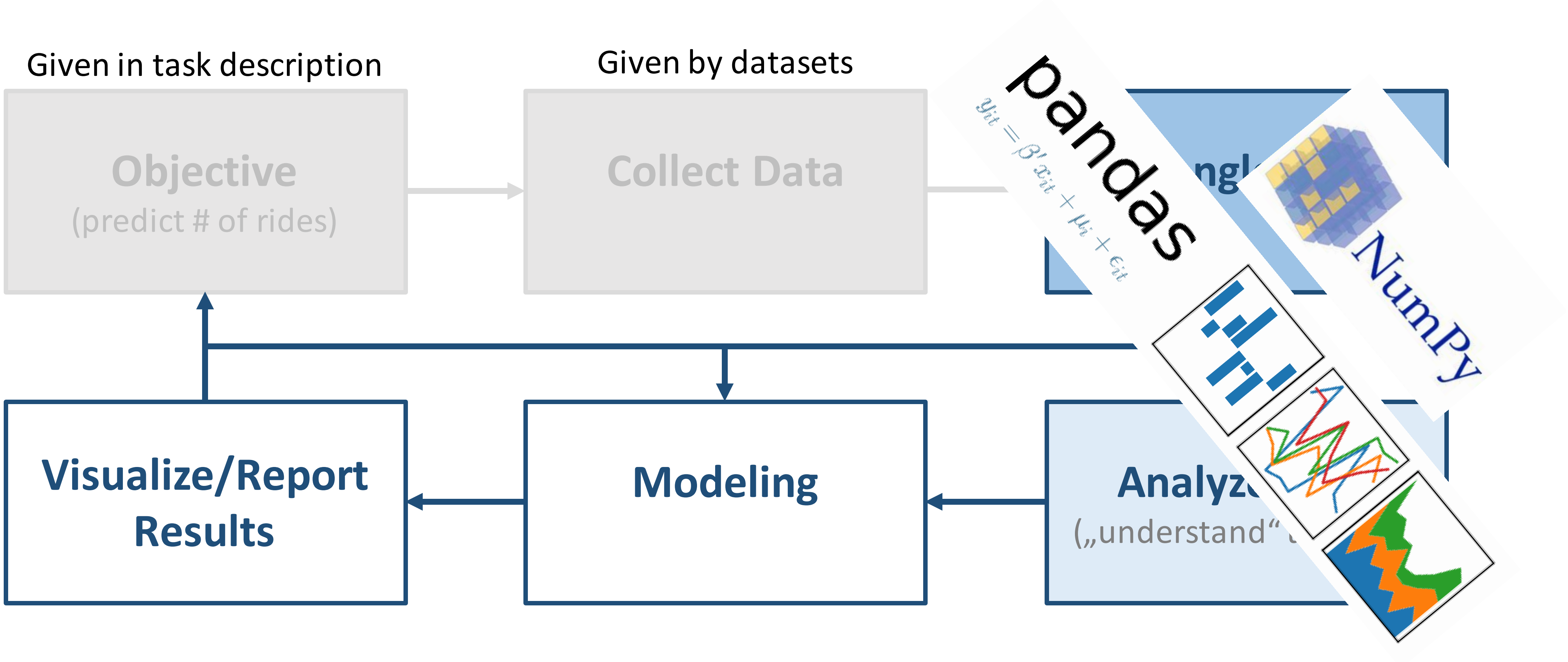
Pandas Recap

- The previous slides hold almost everything you need for
  - Handling missing data (e.g., <filter> == NaN)
  - Basic data analysis
  - Feature engineering

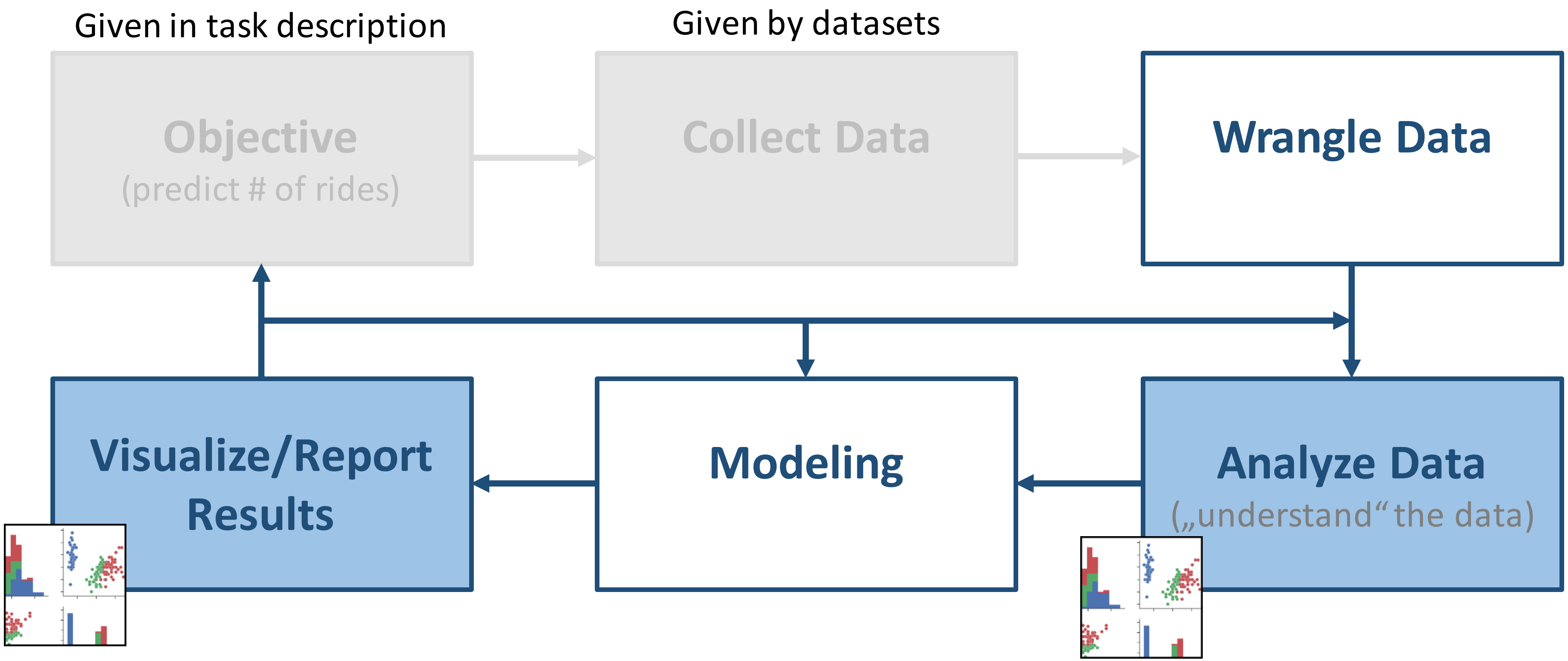
In [22]:
```
def peak_hour(x):
    if (x == 8) or (x >= 16 and x <= 19):
        return 1;
    else:
        return 0

hour_data['peak'] = hour_data['hr'].apply(lambda x: peak_hour(x))
peak_mask = (hour_data['workingday'] == 0) & (hour_data['peak'] == 1)
hour_data.ix[peak_mask, 'peak'] = 0
```

- You can not use pandas for
  - Visualization other than tables and numbers
  - Predictive modeling

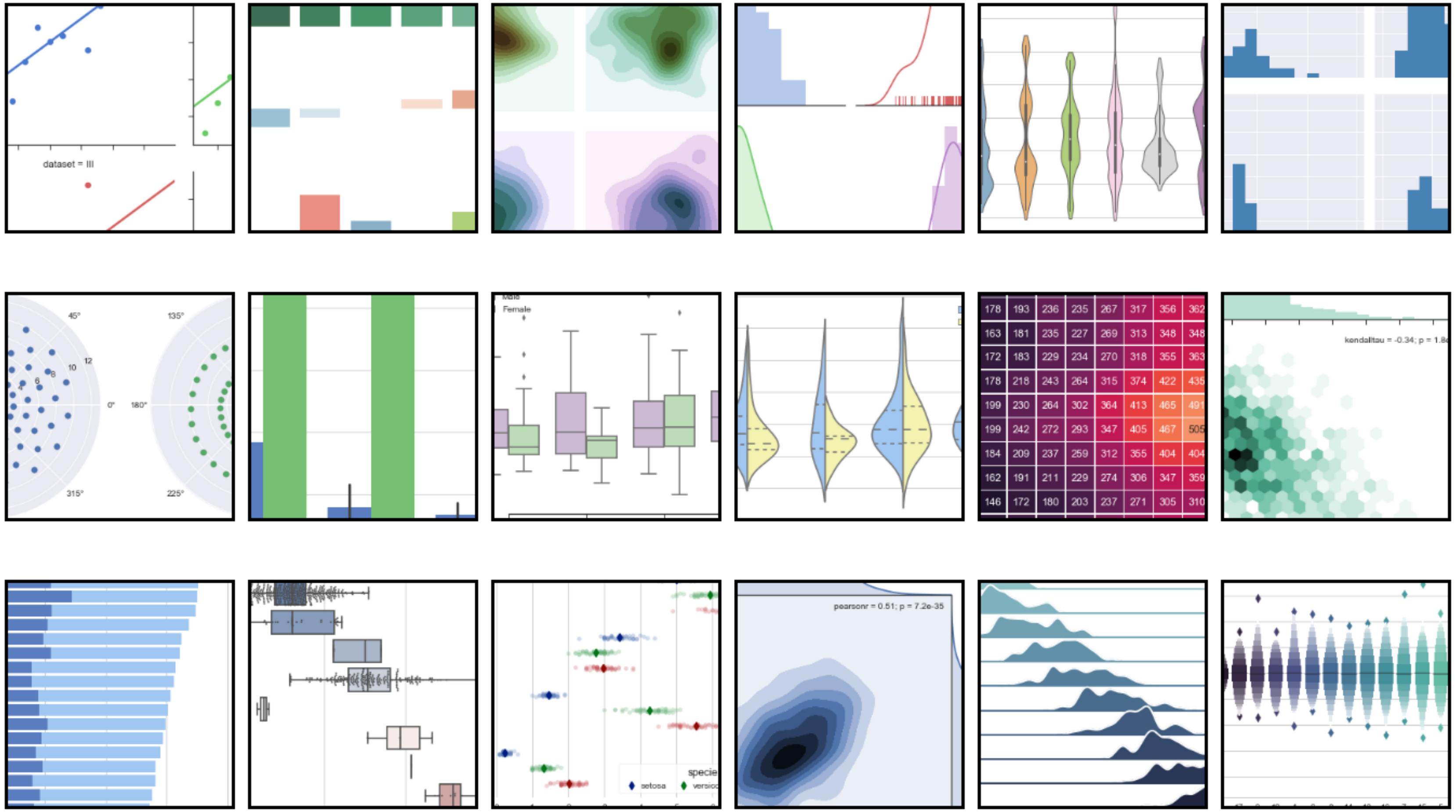
The Python Stack



The Python Stack



Seaborn



https://seaborn.pydata.org/examples/index.html

Seaborn

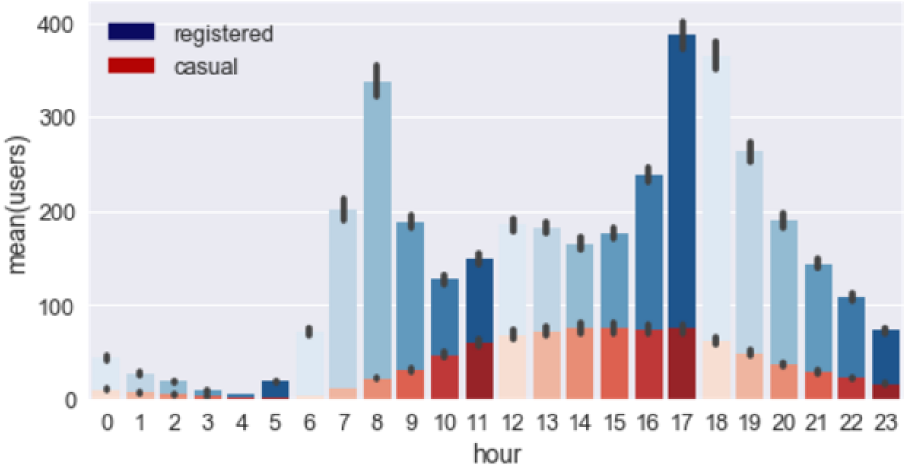
- Visualization library
  - Used for data analysis that goes beyond texts and tables
- Built on top of Matplotlib
  - Customized themes and high-level interface to control Matplotlib figure aesthetics
- Can easily plot distributions, split data on categories, etc.
- First step: `import seaborn as sns`

Seaborn - Examples

- Barplot (e.g., for categorical variables)

In [33]:
```
import matplotlib.patches as mpatches

f, ax = plt.subplots(figsize=(8, 4))
sns.barplot(x=hour_data['hr'], y=hour_data['registered'], palette=sns.color_palette("Blues"));
sns.barplot(x=hour_data['hr'], y=hour_data['casual'], palette=sns.color_palette("Reds"));
ax.set_xlabel('hour')
ax.set_ylabel('mean(users)')

reg_patch = mpatches.Patch(color='#0B0B61', label='registered')
cas_patch = mpatches.Patch(color='#B40404', label='casual')
ax.legend(handles=[reg_patch, cas_patch])
```

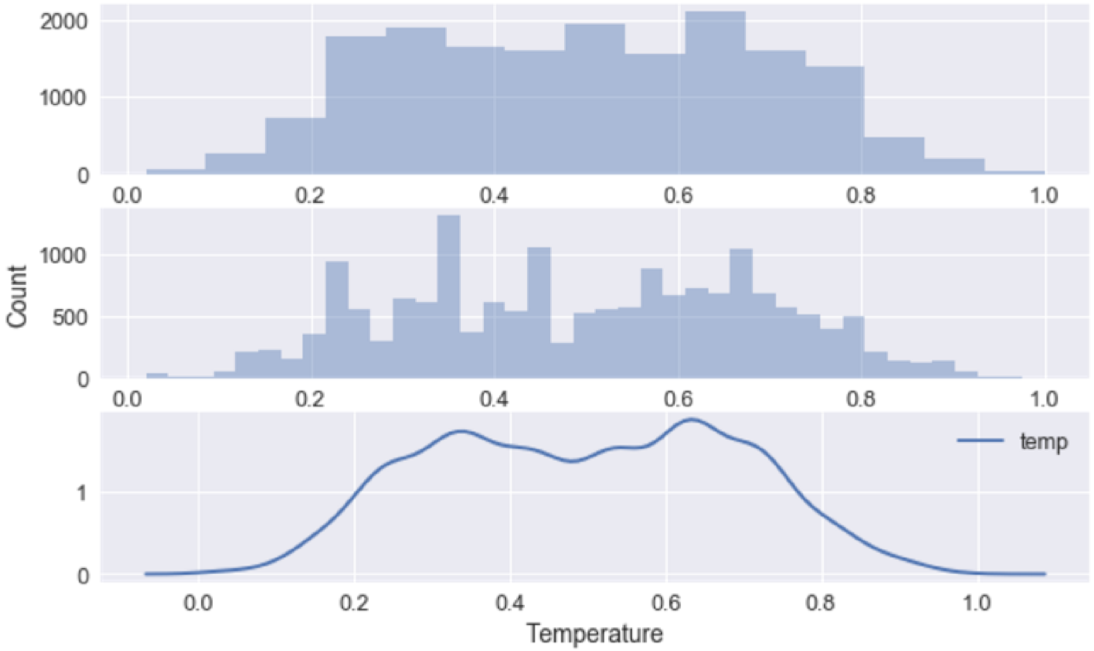
Out[33]: <matplotlib.legend.Legend at 0x14110fd0>



Seaborn - Examples

- Distplot (for investigating distributions)

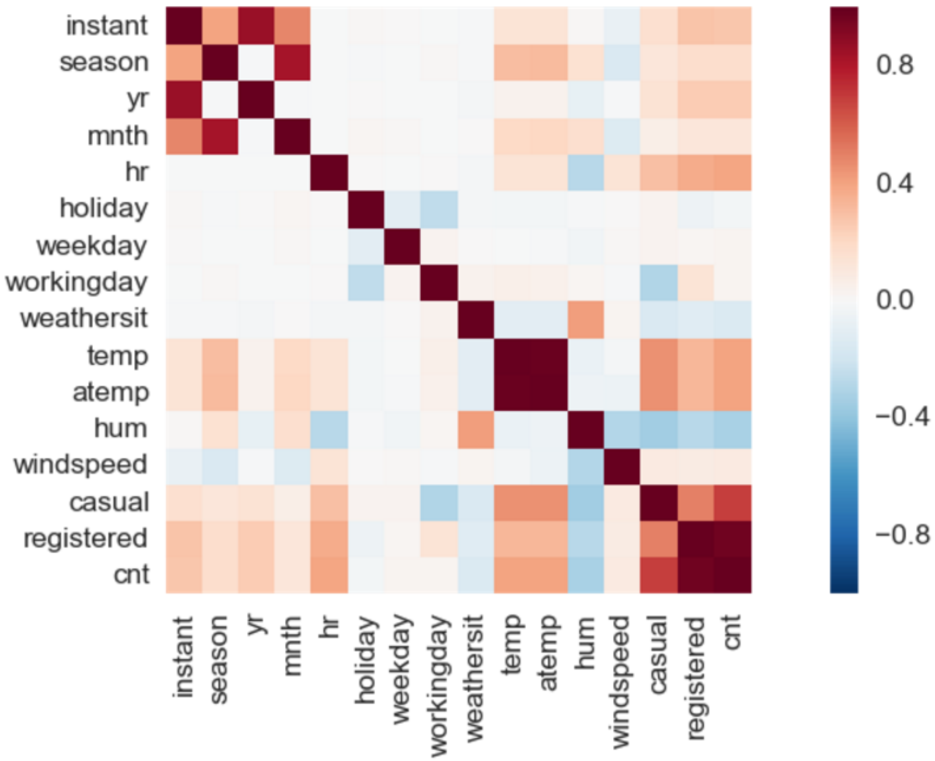
```
In [65]: f, ax = plt.subplots(3,figsize=(10,6))
         sns.distplot(hour_data['temp'], bins=15, kde=False, ax=ax[0])
         sns.distplot(hour_data['temp'], kde=False, ax=ax[1])
         sns.kdeplot(hour_data['temp'], ax=ax[2])
         ax[2].set_xlabel('Temperature')
         ax[1].set_ylabel('Count')

Out[65]: <matplotlib.text.Text at 0x1d3b24e0>
```



Seaborn - Examples

- Correlation matrix

In [8]:
```
# get correlation matrix and visualize it in figure
cmat = hour_data.corr()

f, ax = plt.subplots(figsize=(10, 5))
sns.heatmap(cmat, square=True)
f.tight_layout()
```

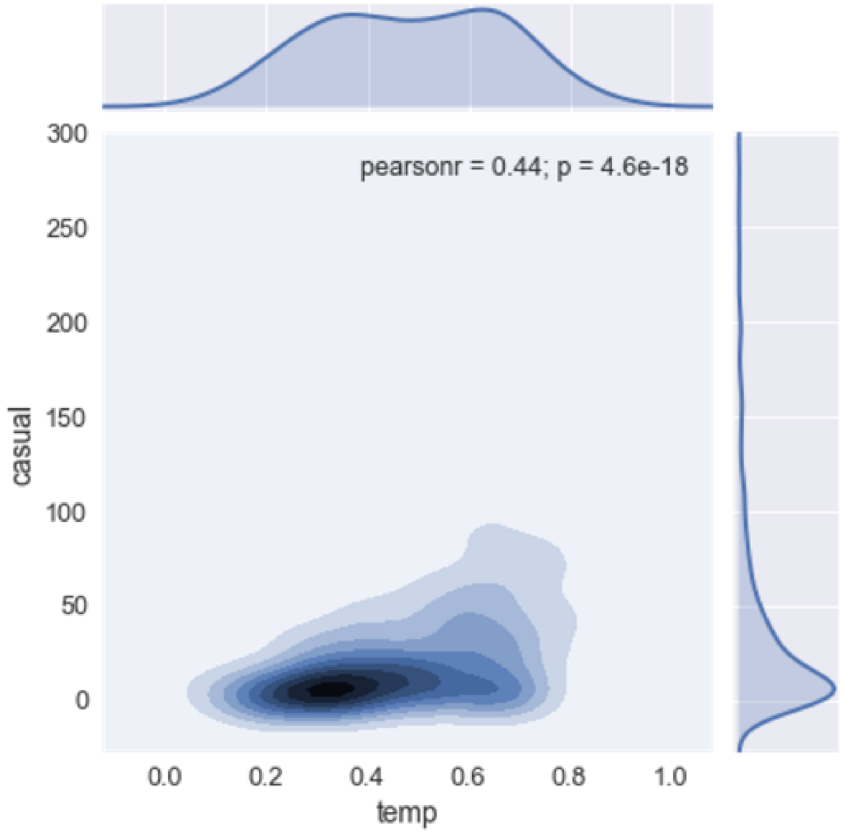


Seaborn - Examples

- Jointplot: investigate bivariate distributions while also retrieving univariate distribution of each variable / feature

In [102]:
```
hr_smpl = hour_data.sample(frac=0.02)
sns.jointplot(x="temp", y="casual", data=hr_smpl, kind='kde')
```

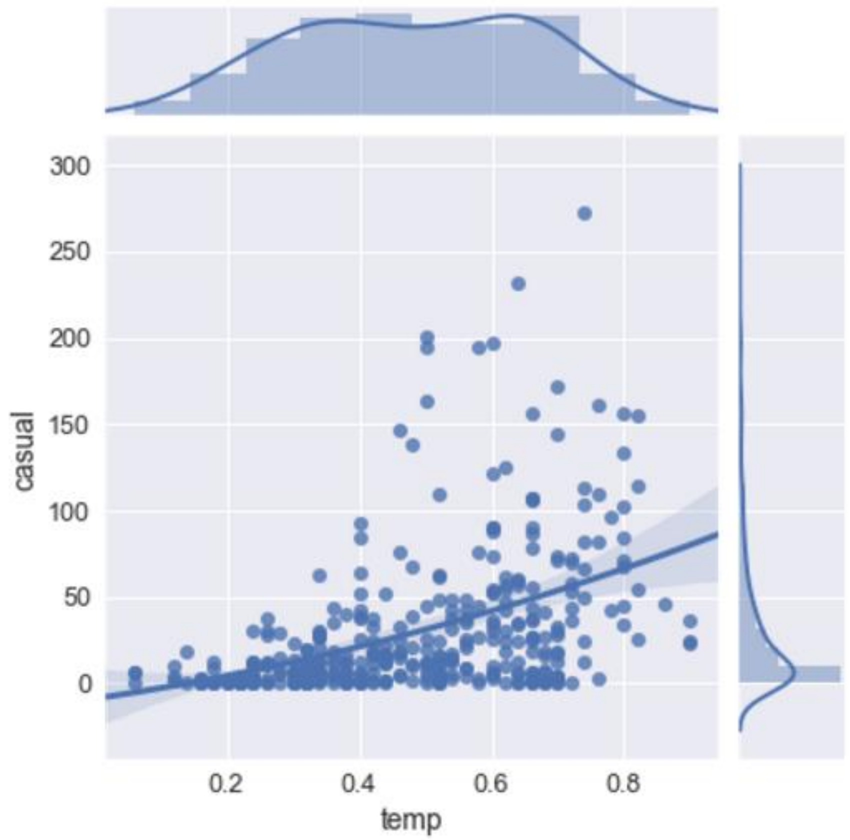
Out[102]: `<seaborn.axisgrid.JointGrid at 0x7bb35940>`



Seaborn - Examples

- Jointplot: investigating bivariate distributions
- JointGrid: slightly more powerful
  - E.g., can fit regression to joint distribution

```
In [103]: g = sns.JointGrid(x="temp", y="casual", data=hr_smpl)
          g = g.plot_joint(sns.regplot, order=2)
          g = g.plot_marginals(sns.distplot)
```

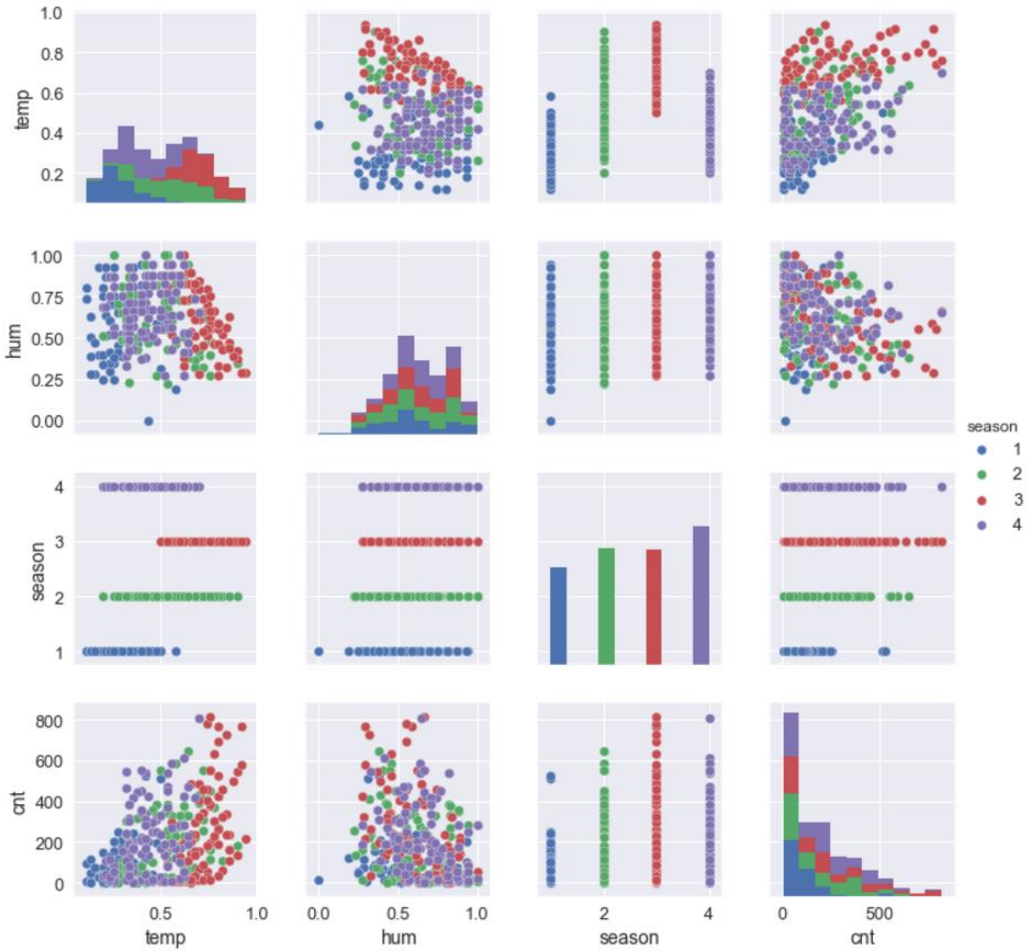


Seaborn - Examples

- Pairplot: investigating bivariate distributions

- **hue** keyword allows to separate data items by a categorical variable
  - Can facilitate discoveries
  - Obviously: higher temp in summer, more rides in summer, etc.

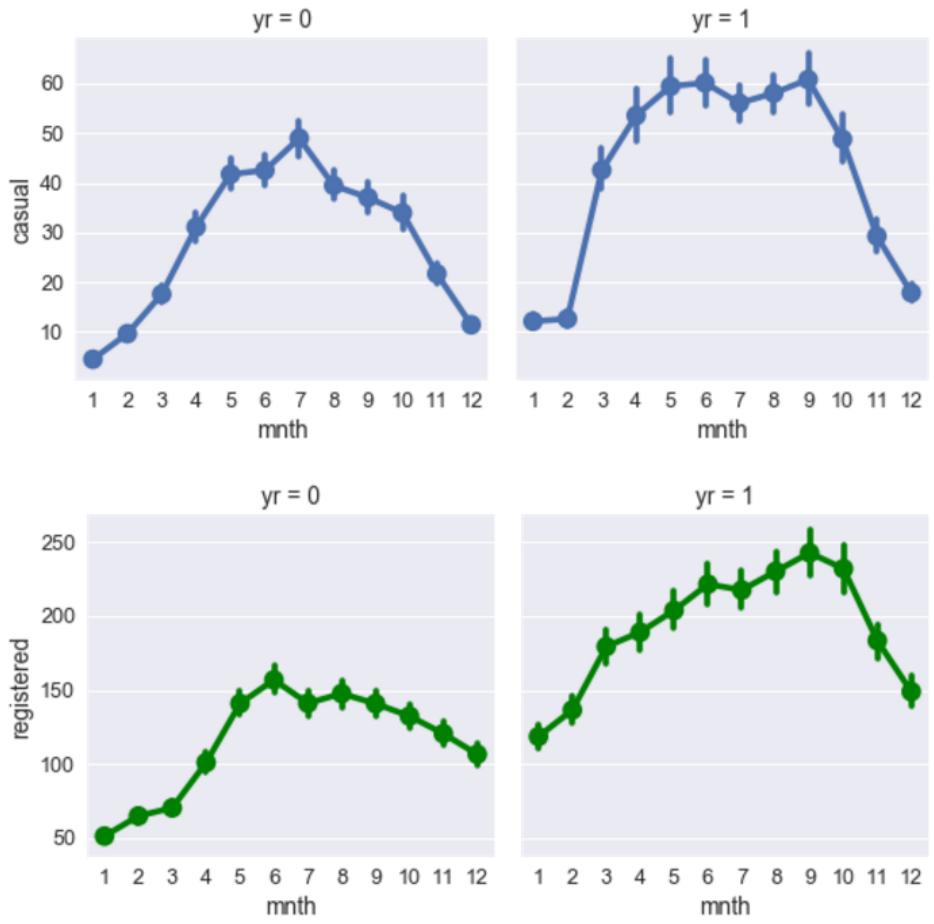
```
In [81]: hr_smpl = hour_data.sample(frac=0.02)
         columns= ['temp','hum','season','cnt']
         sns.pairplot(hr_smpl[columns], hue='season')

Out[81]: <seaborn.axisgrid.PairGrid at 0x6504db70>
```



Seaborn - Examples

- Conditional plots:
  FactorPlot and FacetGrid
  - Allow to segment data based on categorical variables (**col** keyword)

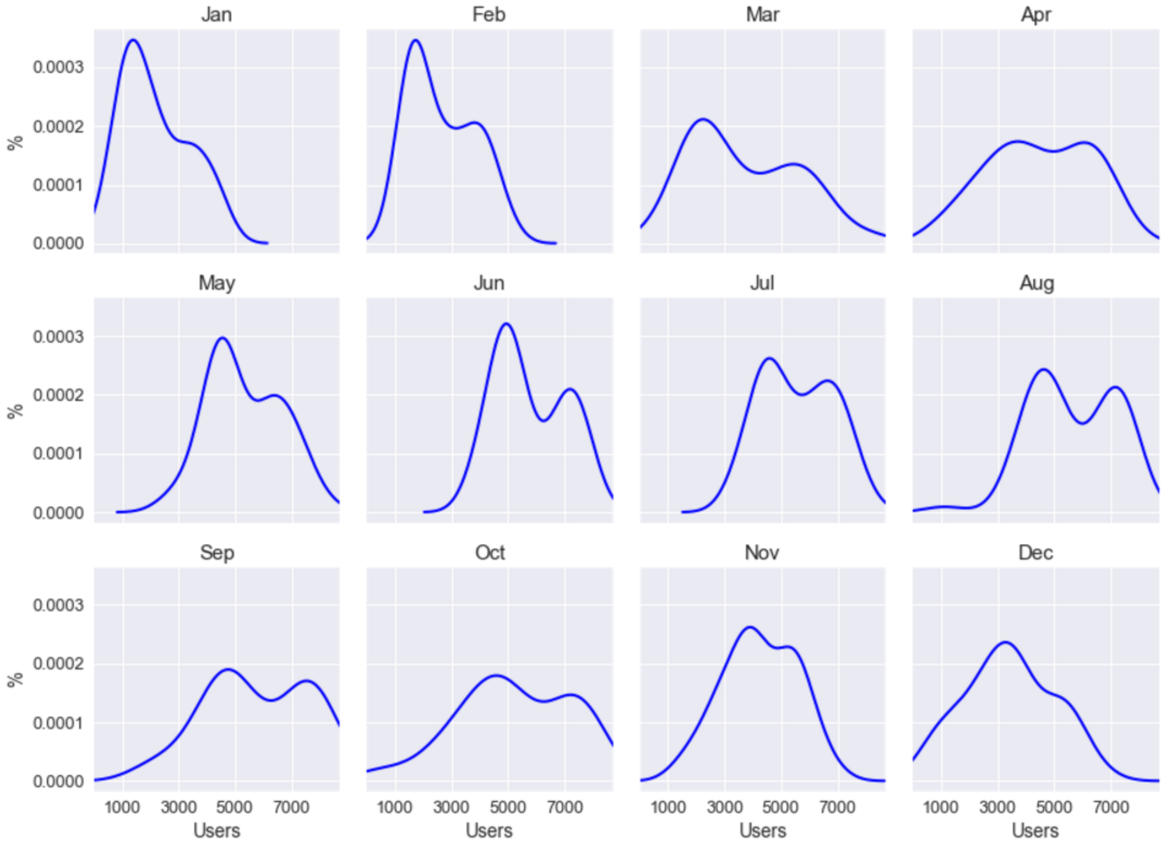
```
In [89]: sns.factorplot(data=hour_data, x="mnth", y="casual", col='yr')
         sns.factorplot(data=hour_data, x="mnth", y="registered", col='yr', color='g')

Out[89]: <seaborn.axisgrid.FacetGrid at 0x75712908>
```



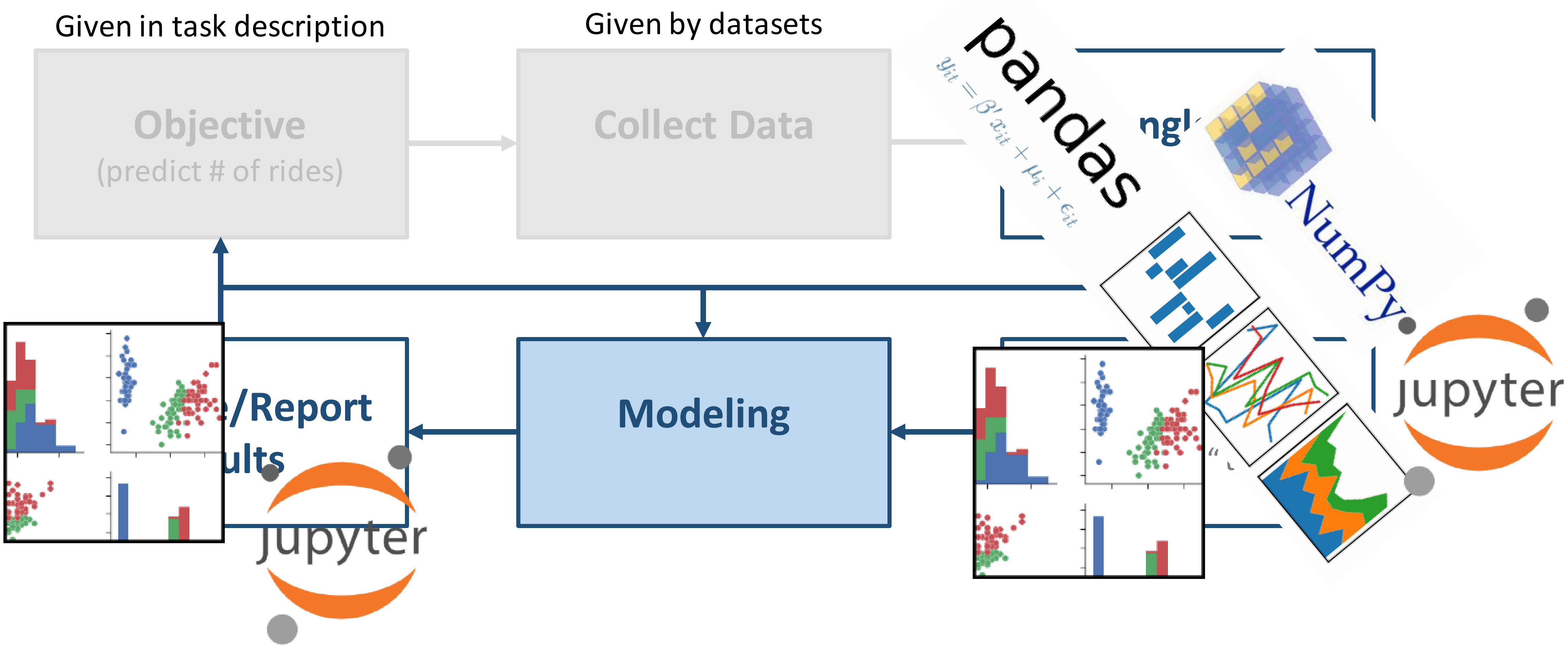
Seaborn - Examples

- Conditional plots: FactorPlot and FacetGrid
  - Allow to segment data based on categorical variables (**col** keyword)

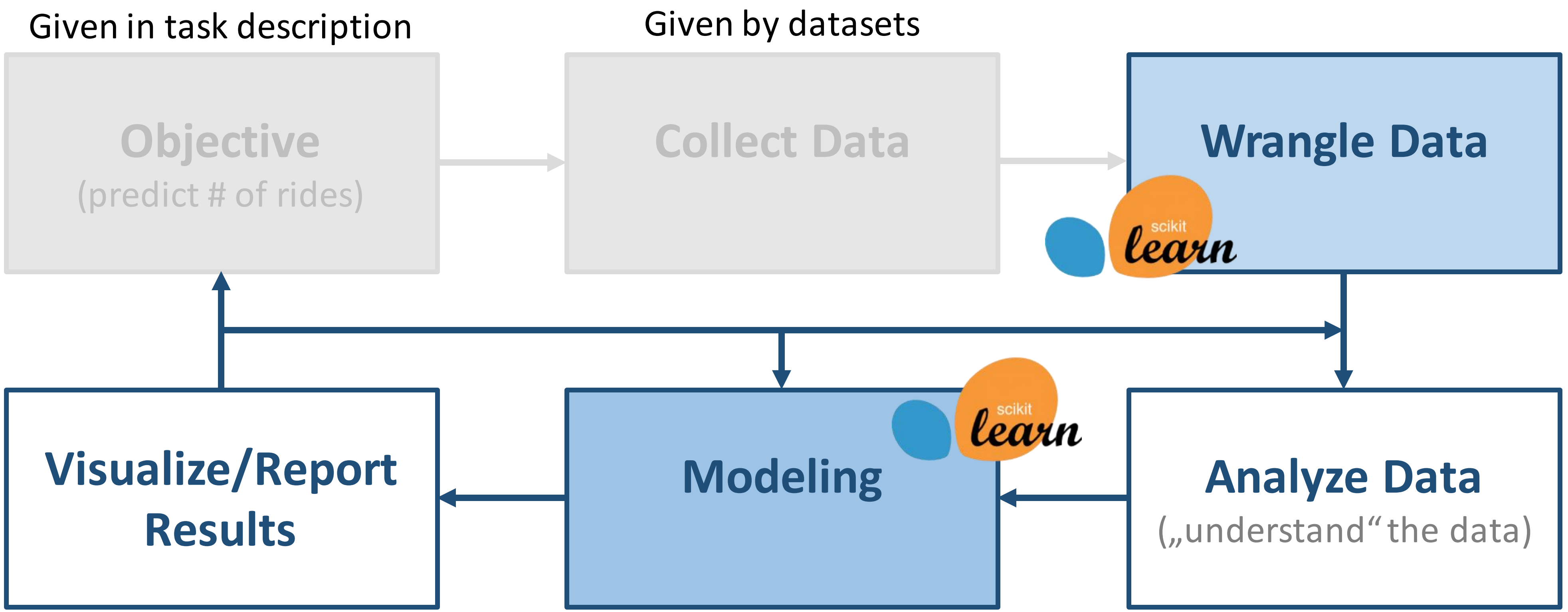
In [36]:
```
g = sns.FacetGrid(day_data, col="mnth", col_wrap=4, size=3, xlim=(0, max(day_data['cnt'])))
g.map(sns.kdeplot, "cnt", color='b');

titles = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
for ax, title in zip(g.axes.flat, titles):
    ax.set_title(title)

g.set_axis_labels("Users", "%");
g.set(xticks=[1000, 3000, 5000, 7000]);
```



The Python Stack



The Python Stack



SciKit Learn

- Python library for predictive modeling

- Offers large range of machine learning algorithm implementations
  - Easily accessible via API

- Besides, built-in functions for a large range of related tasks
  - E.g., functions for test-train split or cross validation
  - Functions for data preprocessing

A few words on SciKit Learn in this Course

- Learning by Doing!
- We don‘t have time to walk through each and every algorithm and function
  - >70 algorithms implemented for supervised learning alone
- Try yourself ☺

SciKit Learn: Preprocessing

- Useful methods for imputation and normalization

In [33]:
```
from sklearn.preprocessing import Imputer
df = pd.DataFrame([[1,2,3],[20,21,10],[1, None, 3],[2,2,4]])
```

In [34]:
```
df
```

Out[34]:

	0	1	2
0	1	2.0	3
1	20	21.0	10
2	1	NaN	3
3	2	2.0	4

In [35]:
```
imp = Imputer(missing_values='NaN', strategy='most_frequent', axis=0)
imp.fit(df)
df = imp.transform(df);
```

In [36]:
```
df
```

Out[36]:
```
array([[  1.,   2.,   3.],
       [ 20.,  21.,  10.],
       [  1.,   2.,   3.],
       [  2.,   2.,   4.]])
```

In [49]:
```
from sklearn.preprocessing import normalize
df = pd.DataFrame([[1,2,3],[20,21,10],[1, 5, 3],[2,2,4]])
```

In [50]:
```
df
```

Out[50]:

	0	1	2
0	1	2	3
1	20	21	10
2	1	5	3
3	2	2	4

In [51]:
```
df = normalize(df, norm='max', axis=0)
```

In [52]:
```
df
```

Out[52]:
```
array([[ 0.05      ,  0.0952381 ,  0.3       ],
       [ 1.        ,  1.        ,  1.        ],
       [ 0.05      ,  0.23809524,  0.3       ],
       [ 0.1       ,  0.0952381 ,  0.4       ]])
```

Recap: Modeling Pipeline

- Remember modeling guide from introduction:

Model Idea -> Algorithm -> Split Data -> Fit Model -> Test on Test Data

- Model Idea: you need to come up with this yourself
  - Type of problem
  - Single vs multiple models
  - Feature engineering
  - …

SciKit Learn: Selection of Algorithm

1. Supervised learning

1.1. Generalized Linear Models

- 1.1.1. Ordinary Least Squares
  - 1.1.1.1. Ordinary Least Squares Complexity
- 1.1.2. Ridge Regression
  - 1.1.2.1. Ridge Complexity
  - 1.1.2.2. Setting the regularization parameter: generalized Cross-Validation
- 1.1.3. Lasso
  - 1.1.3.1. Setting regularization parameter
    - 1.1.3.1.1. Using cross-validation
    - 1.1.3.1.2. Information-criteria based model selection
    - 1.1.3.1.3. Comparison with the regularization parameter of SVM
- 1.1.4. Multi-task Lasso
- 1.1.5. Elastic Net
- 1.1.6. Multi-task Elastic Net
- 1.1.7. Least Angle Regression
- 1.1.8. LARS Lasso
  - 1.1.8.1. Mathematical formulation
- 1.1.9. Orthogonal Matching Pursuit (OMP)
- 1.1.10. Bayesian Regression
  - 1.1.10.1. Bayesian Ridge Regression
  - 1.1.10.2. Automatic Relevance Determination - ARD
- 1.1.11. Logistic regression
- 1.1.12. Stochastic Gradient Descent - SGD
- 1.1.13. Perceptron
- 1.1.14. Passive Aggressive Algorithms
- 1.1.15. Robustness regression: outliers and modeling errors
  - 1.1.15.1. Different scenario and useful concepts
  - 1.1.15.2. RANSAC: RANdom SAmple Consensus
    - 1.1.15.2.1. Details of the algorithm
  - 1.1.15.3. Theil-Sen estimator: generalized-median-based estimator
    - 1.1.15.3.1. Theoretical considerations

1.2. Linear and Quadratic Discriminant Analysis

- 1.2.1. Dimensionality reduction using Linear Discriminant Analysis
- 1.2.2. Mathematical formulation of the LDA and QDA classifiers
- 1.2.3. Mathematical formulation of LDA dimensionality reduction
- 1.2.4. Shrinkage
- 1.2.5. Estimation algorithms

1.3. Kernel ridge regression

1.4. Support Vector Machines

- 1.4.1. Classification
  - 1.4.1.1. Multi-class classification
  - 1.4.1.2. Scores and probabilities
  - 1.4.1.3. Unbalanced problems
- 1.4.2. Regression
- 1.4.3. Density estimation, novelty detection
- 1.4.4. Complexity
- 1.4.5. Tips on Practical Use
- 1.4.6. Kernel functions
  - 1.4.6.1. Custom Kernels
    - 1.4.6.1.1. Using Python functions as kernels
    - 1.4.6.1.2. Using the Gram matrix
    - 1.4.6.1.3. Parameters of the RBF Kernel
- 1.4.7. Mathematical formulation
  - 1.4.7.1. SVC
  - 1.4.7.2. NuSVC
  - 1.4.7.3. SVR
- 1.4.8. Implementation details

1.5. Stochastic Gradient Descent

- 1.5.1. Classification
- 1.5.2. Regression
- 1.5.3. Stochastic Gradient Descent for sparse data
- 1.5.4. Complexity
- 1.5.5. Tips on Practical Use
- 1.5.6. Mathematical formulation
  - 1.5.6.1. SGD

1.10. Decision Trees

- 1.10.1. Classification
- 1.10.2. Regression
- 1.10.3. Multi-output problems
- 1.10.4. Complexity
- 1.10.5. Tips on practical use
- 1.10.6. Tree algorithms: ID3, C4.5, C5.0 and CART
- 1.10.7. Mathematical formulation
  - 1.10.7.1. Classification criteria
  - 1.10.7.2. Regression criteria

1.11. Ensemble methods

- 1.11.1. Bagging meta-estimator
- 1.11.2. Forests of randomized trees
  - 1.11.2.1. Random Forests
  - 1.11.2.2. Extremely Randomized Trees
  - 1.11.2.3. Parameters
  - 1.11.2.4. Parallelization
  - 1.11.2.5. Feature importance evaluation
  - 1.11.2.6. Totally Random Trees Embedding
- 1.11.3. AdaBoost
  - 1.11.3.1. Usage
- 1.11.4. Gradient Tree Boosting
  - 1.11.4.1. Classification
  - 1.11.4.2. Regression
  - 1.11.4.3. Fitting additional weak-learners
  - 1.11.4.4. Controlling the tree size
  - 1.11.4.5. Mathematical formulation
    - 1.11.4.5.1. Loss Functions
  - 1.11.4.6. Regularization
    - 1.11.4.6.1. Shrinkage
    - 1.11.4.6.2. Subsampling
  - 1.11.4.7. Interpretation
    - 1.11.4.7.1. Feature importance
    - 1.11.4.7.2. Partial dependence
- 1.11.5. Voting Classifier
  - 1.11.5.1. Majority Class Labels (Majority/Hard Voting)
    - 1.11.5.1.1. Usage

Train/Test Split

- Why do we need to split into train and test data?
  - The goal of every model in data science is:

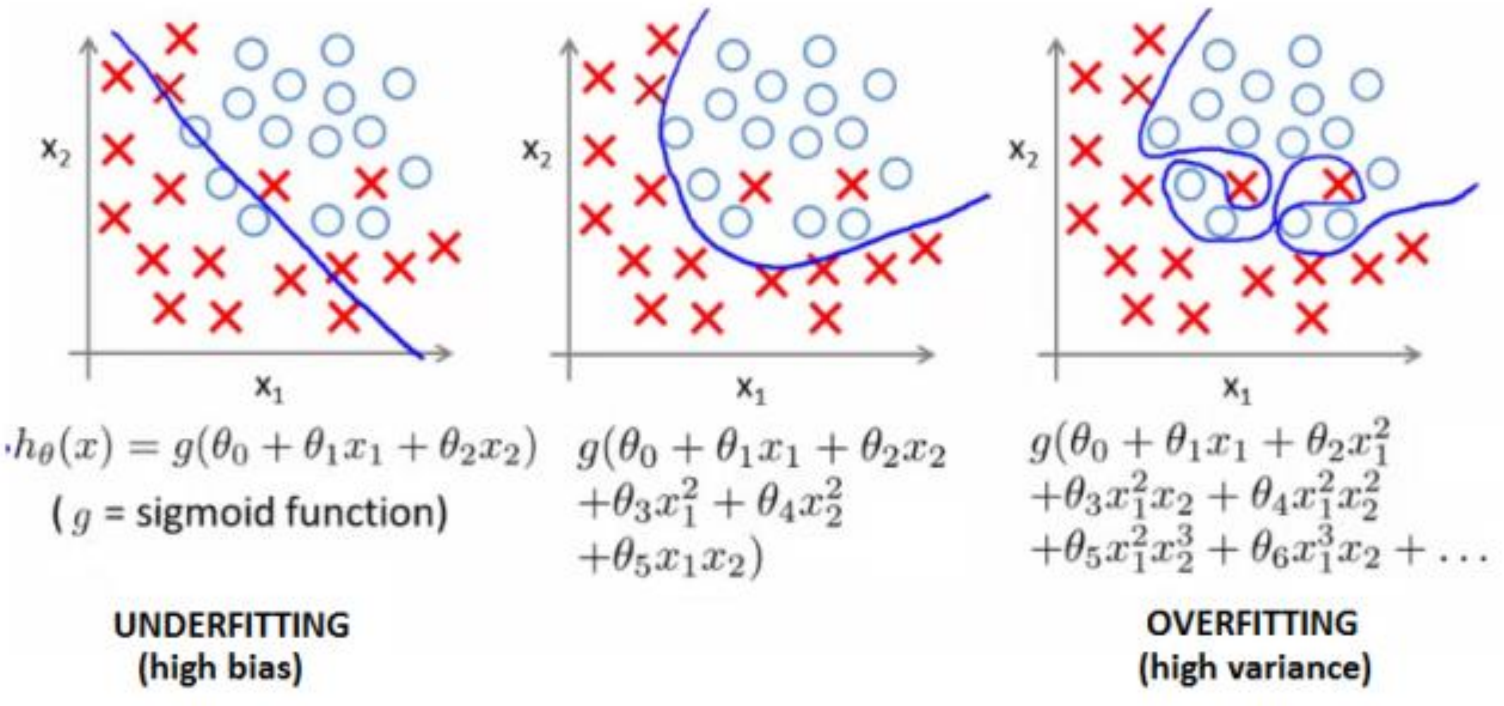
Predicting previously unseen data points

- What would happen if we only use a single data set?

Overfitting!

- Model would perfectly learn structure of data, but would not generalize

Train/Test Split



Taken from mlwiki.org

Train/Test Split

- How do we split data?
  - Rule of thumb: 90/10 to 70/30 (depends on the amount of data you have)

- If enough data: consider an additional validation/hold-out set
  - E.g., 70-20-10 on train/test/validation
  - Fit model on train
  - Check for improvement in validation
  - Only rarely touch test set to verify improvements

- Another option: cross-validation (see later lectures)

SciKit Learn: Train/Test Split

- How to split data?

```
In [24]: from sklearn.model_selection import train_test_split
         train, test = train_test_split(all_data, test_size=0.1)
```

- For validation set: do the same thing recursively on train set

Fitting the Model

- Once you have decided: easy, few lines of code
- Series of steps:
  - Import correct module
  - Instantiate model object (optional: with parameters; see later lectures)
  - Fit model

```
In [29]: from sklearn.linear_model import LinearRegression
         lr = LinearRegression()
         lr.fit(train,target_train_registered)
```

Predictions & Testing the Model

- Prediction itself: just one line of code

```
In [ ]: predictions = lr.predict(test)
```

- Evaluation of the model:
  - Quality of predictions (wrt. previously defined metric):

```
print 'MAE: ', mean_absolute_error(target_test_registered, predictions)
```

```
MAE:  71.5744611842
```

Predictions & Testing the Model

- Evaluation of the model:
  - Usefulness of features (note: shown here on RandomForest):

```
In [35]: print train.columns
         rfr.feature_importances_

Index([u'instant', u'season', u'yr', u'mnth', u'hr', u'holiday', u'weekday',
       u'workingday', u'weathersit', u'temp', u'atemp_x', u'hum_x',
       u'windspeed_x', u'peak', u'atemp_y', u'hum_y', u'windspeed_y'],
      dtype='object')

Out[35]: array([ 2.05540749e-01,  1.08326368e-03,  1.07021487e-04,
                 2.86040450e-03,  2.73137846e-01,  1.22699884e-03,
                 9.15607940e-03,  3.25442302e-02,  1.87012402e-02,
                 9.76701103e-03,  1.15701744e-02,  1.26128059e-02,
                 4.46063084e-03,  3.84129829e-01,  1.72215955e-02,
                 7.98755358e-03,  7.89256619e-03])
```

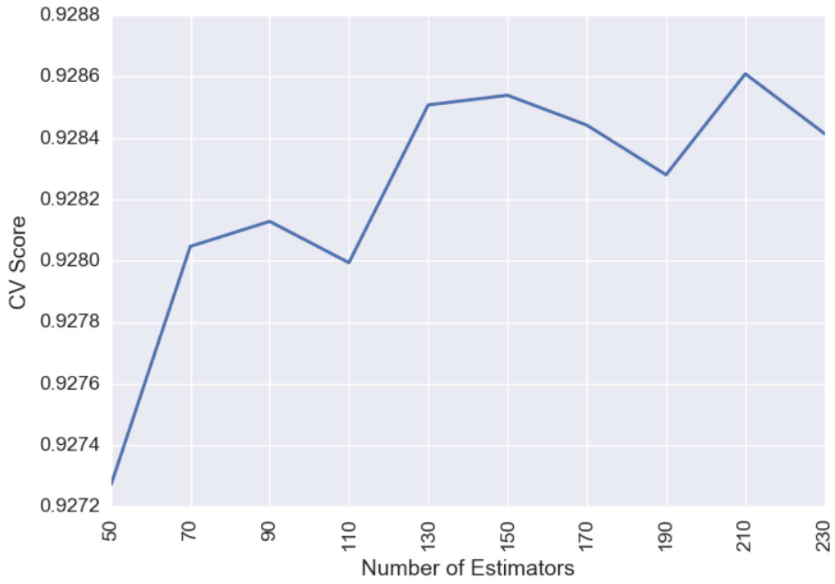
  - Note: Insights of such analysis can be used for feature selection (see later lectures)

Predictions & Testing the Model

- Evaluation of the model:
  - Parameter Tuning (details: see last lecture):

In [36]:
```
cv_score = []
for i in np.arange(50, 250, 20):
    rfr = RandomForestRegressor(n_estimators=i)
    cv_score.append(np.mean(cross_val_score(rfr, train, target_train_casual, cv=5)))
```

In [37]:
```
plt.plot(cv_score)
plt.ylabel('CV Score')
plt.xlabel('Number of Estimators');
plt.xticks(np.arange(0,10,1), np.arange(50, 250, 20), rotation='vertical');
```



Summary

- Today we discussed the Python DS stack
- Obviously: a lot of content

- This lecture should serve as an indication about
  - what tools and libraries you can use for investigating and manipulating data and
  - how to use them effectively
  - what tools and libraries you can use for modeling

- This lecture is not intended for you to now know everything there is ☺

Summary

- You will get a lot of exercise in the practical tasks

- We will look at some algorithmic libraries and model validation techniques in more detail later

- Theoretical knowledge on algorithms: you hopefully have it from previous courses