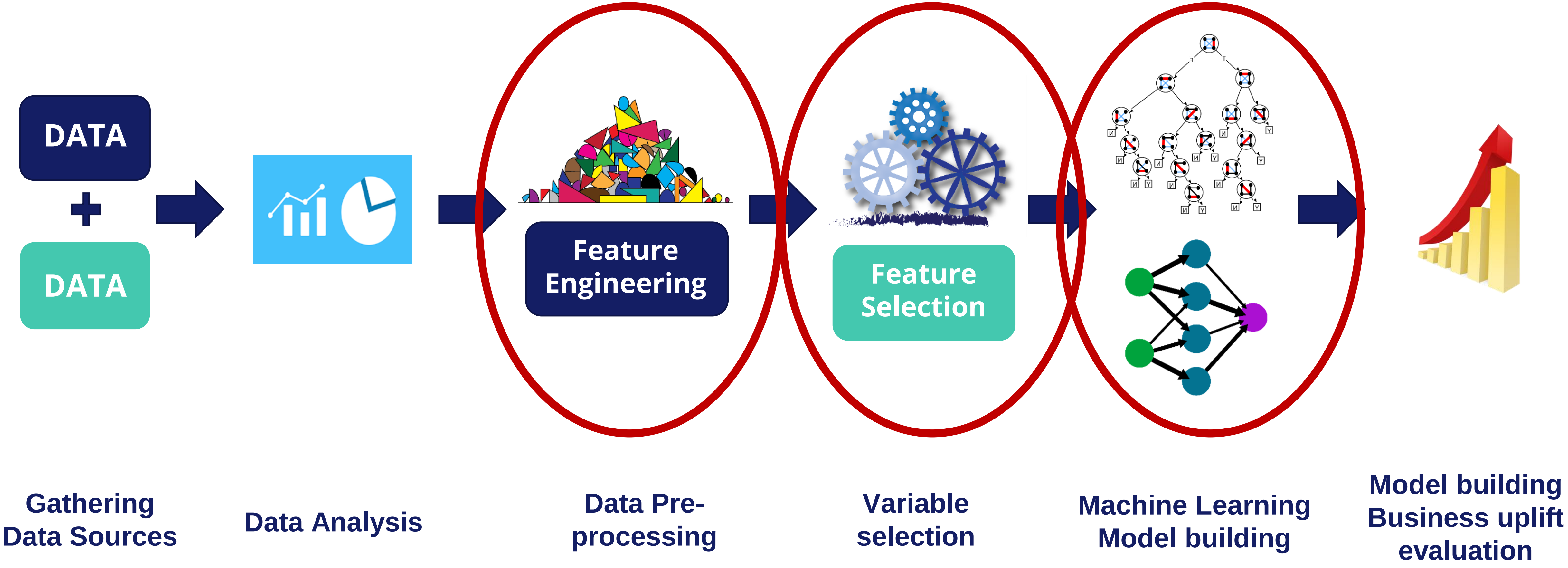
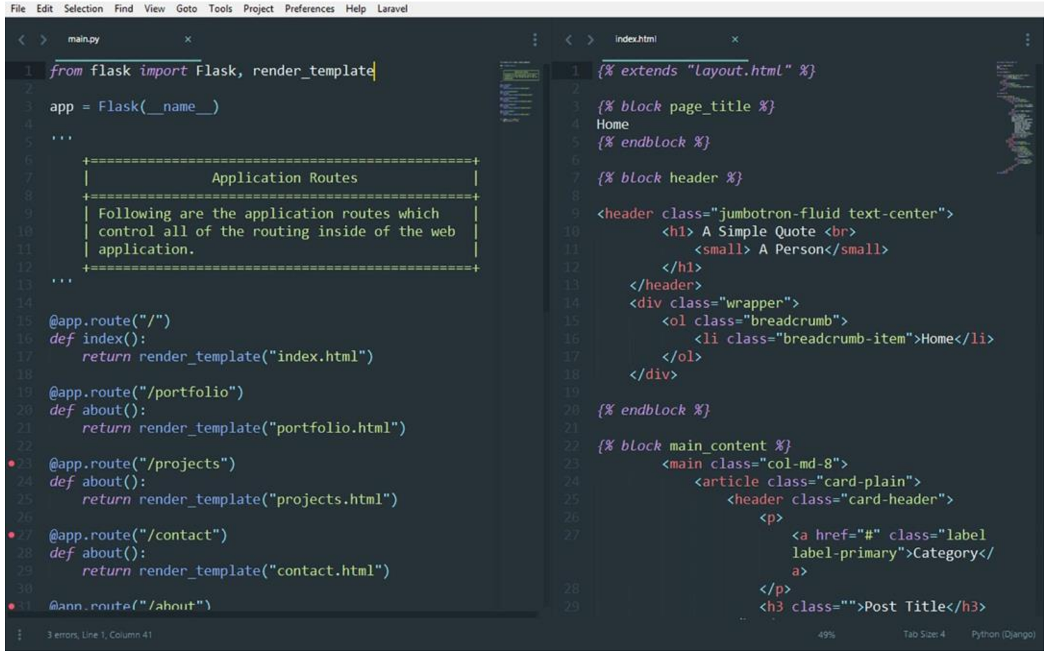
Writing Production Code for Machine Learning Deployment

Overview

Machine Learning Pipeline: Production



Towards deployment code



Code to:

- Create and Transform features
- Incorporate the feature selection
- Build Machine Learning Models
- Score new data

How to write deployment code for ML

- Procedural Programming
- Custom Pipeline Code
- Third Party Pipeline Code

Procedural Programming

Procedural Programming

In procedural programming, procedures, also known as routines, subroutines or functions, are carried out as a series of computational steps.

For us, it refers to writing the series of feature creation, feature transformation, model training and data scoring steps, as functions, that then we can call and run one after the other.

Procedural Programming: the functions

```python
# ==== EXAMPLE OF PROCEDURAL PROGRAMMING SCRIPT ===

import pandas as pd
import numpy as np

# to divide train and test set
from sklearn.model_selection import train_test_split

# feature scaling
from sklearn.preprocessing import StandardScaler

# to build the models
from sklearn.linear_model import LinearRegression, Lasso

# to evaluate the models
from sklearn.metrics import mean_squared_error


# Functional programming, getting all parameters from yaml file

def load_data(df_path):
    return pd.read_csv(df_path)


def divide_train_test(df, target):
    X_train, X_test, y_train, y_test = train_test_split(df, df[target],
                                                        test_size=0.2,
                                                        random_state=0)
    return X_train, X_test, y_train, y_test


def remove_numerical_na(df, var, mean_val):
    return df[var].fillna(mean_val)


def remove_categorical_na(df, var):
    return df[var].fillna('Missing')


def cap_outliers(df, var, cap, bigger_than=False):
    if bigger_than:
        capped_var = np.where(df[var]>cap, cap, df[var])
    else:
        capped_var = np.where(df[var]<cap, cap, df[var])

    return capped_var
```


```python
def remove_numerical_na(df, var, mean_val):
    return df[var].fillna(mean_val)


def remove_categorical_na(df, var):
    return df[var].fillna('Missing')


def cap_outliers(df, var, cap, bigger_than=False):
    if bigger_than:
        capped_var = np.where(df[var]>cap, cap, df[var])
    else:
        capped_var = np.where(df[var]<cap, cap, df[var])

    return capped_var
```


```python


def transform_skewed_variables(df, var):
    return np.log(df[var])


def remove_rare_labels(df, var, frequent_labels):
    return np.where(df[var].isin(frequent_labels, df[var], 'Rare'))


def train_scaler(df, output_path):
    scaler = StandardScaler()
    scaler.fit(df)
    joblib.save(scaler, output_path)
    return scaler

def scale_features(df, scaler):
    scaler = load(scaler) # with joblib probably
    return scaler.transform(df)


def train_model(df, target, features, scaler, output_path):
    lin_model = Lasso(random_state=2909)
    lin_model.fit(scaler.transform(df[features]), target)
    joblib.save(lin_model, output_path)
    return lin_model

def predict(df, model, features, scaler):
    return model.predict_proba(scaler.transform(df[features]))
```

The functions or procedures to create and transform features, and to train and save the models and make the predictions

Procedural Programming: train script

```
#=========== training pipeline ======

df = load(yaml_path_to_file)
train, test, y_train, y_test = divide_train_test(df, yaml_target_name)

# remove NA numerical
train[var1] = remove_numerical_na(train, var1, mean_val1_in_yaml)
train[var2] = remove_numerical_na(train, var2, mean_val2_in_yaml)

train[var3] = remove_categorical_na(train[var3])
train[var4] = remove_categorical_na(train[var4])

train[var5] = cap_outliers(train, var5, cap_value_in_yaml, bigger_than=False)
train[var6] = cap_outliers(train, var6, cap_value_in_yaml, bigger_than=False)

train[var7] = transform_skewed_variables(train, var7)

train[var8] = remove_rare_labels(train, var8, frequent_labels_in_yaml)

scaler = train_scaler(train, output_path_in_yaml)

lin_model = train_model(train, y_train, feature_list_in_yaml, scaler, output_path_in_yaml)

#== END
```

Calls the previous functions in order, to train and save the models

Procedural Programming: score script

```python
# =========== scoring pipeline =========

data = 'load it from somewhere'

# remove NA numerical
data[var1] = remove_numerical_na(data, var1, mean_val1_in_yaml)
data[var2] = remove_numerical_na(data, var2, mean_val2_in_yaml)

data[var3] = remove_categorical_na(data[var3])
data[var4] = remove_categorical_na(data[var4])

data[var5] = cap_outliers(data, var5, cap_value_in_yaml, bigger_than=False)
data[var6] = cap_outliers(data, var6, cap_value_in_yaml, bigger_than=False)

data[var7] = transform_skewed_variables(data, var7)

data[var8] = remove_rare_labels(data, var8, frequent_labels_in_yaml)

scaler = joblib.load((output_path_in_yaml_to_scaler)
lin_model = joblib.load(output_path_in_yaml_to_model)

score = predict(data, lin_model, feature_list_in_yaml, scaler)

# ===== END
```

Calls the previous functions in order, to score new data

Procedural Programming: yaml file

```
### example of yaml ####

#paths
path_to_dataset = "path_to_my_dataset"
output_scaler_path = 'path_to_store_scaler'
output_model_path = 'path_to_store_model'

# preproc
var1_mean_val = 1
var2_mean_val = 2

var4_cap_value = 1000
var5_cap_value = 5000

var8_frequent_labels = ['frequent1', 'frequent2', 'frequent3']

# features
features = ['var1', 'var2', 'var3', 'var4', 'etc']


#====== END =======
```

Hard coded variables to engineer, and values to use to transform features.

Hardcoded paths to retrieve and store data

By changing these values, we can re-adjust our models

Procedural Programming: Overview

Advantages

- Straightforward from jupyter notebook
- No software development skills required
- Easy to manually check if it reproduces the original model

Disadvantages

- Can get buggy
- Difficult to test
- Difficult to build software on top of it
- Need to save a lot of intermediate files to store the transformation parameters

Custom Machine Learning Pipeline

Custom ML Pipeline: OOP

In **Object-oriented programming** (**OOP**) we write code in the form of “objects”.

This “objects” can store **data**, and can also store instructions or **procedures** to modify that data.

- Data ⇒ attributes
- Instructions or procedures ⇒ methods

Custom ML Pipeline: Pipeline

A **pipeline** is a set of data processing steps connected in series, where typically, the output of one element is the input of the next one.

The elements of a pipeline can be executed in parallel or in time-sliced fashion. This is useful when we require use of big data, or high computing power, e.g., for neural networks.

Custom ML Pipeline: Summary

A custom Machine Learning pipeline is therefore a sequence of steps, aimed at loading and transforming the data, to get it ready for training or scoring, where:

- We write the processing steps as objects (OOP)
- We write the sequence, i.e., the pipeline as objects (OOP)

Custom ML Pipeline: Example

Download the attached resources as an example.

Custom Pipeline: Overview

Advantages

- Can be tested, versioned, tracked and controlled
- Can build future models on top
- Good software developer practice
- Built to satisfy business needs

Disadvantages

- Requires team of software developers to build and maintain
- Overhead for DS to familiarise with code for debugging or adding on future models
- Preprocessor not reusable, need to re-write Preprocessor class for each new ML model
- Need to write new pipeline for each new ML model
- Lacks versatility, may constrain DS to what is available with the implemented pipeline

Third Party Machine Learning Pipeline: Leveraging the power of Scikit-Learn

Scikit-Learn and sklearn pipeline

Scikit-Learn is a Python library that provides a solid implementation of a range of machine learning algorithms.

Scikit-Learn provides efficient versions of a large number of common algorithms.

Scikit-Learn is characterised by a clean, uniform, and streamlined API.

Scikit-Learn is written so that most of its algorithms follow the same functionality

Once you understand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward

Scikit-Learn provides useful and complete online documentation that allows you to understand both what the algorithm is about and how to use it from scikit-learn

Scikit-Learn is so well established in the community, that new packages are typically designed following scikit-learn functionality to be quickly adopted by end users, e.g, Keras, MLXtend.

Scikit-Learn and sklearn pipeline

Scikit-Learn Objects

- **Transformers** - class that have fit and transform method, it transforms data
    - Scalers
    - Feature selectors
    - One hot encoders.
- **Predictor** - class that has fit and predict methods, it fits and predicts.
    - Any ML algorithm like lasso, decision trees, svm, etc
- **Pipeline** - class that allows you to list an run transformers and predictors in sequence
    - All steps should be transformers except the last one
    - Last step should be a predictor

Scikit-Learn and sklearn pipeline

Here is a good example of Pipeline usage. Pipeline gives you a single interface for all 3 steps of transformation and resulting estimator. It encapsulates transformers and predictors inside, and now you can do something like:

```
vect = CountVectorizer()
tfidf = TfidfTransformer()
clf = SGDClassifier()

vX = vect.fit_transform(Xtrain)
tfidfX = tfidf.fit_transform(vX)
predicted = clf.fit_predict(tfidfX)

# Now evaluate all steps on test set
vX = vect.transform(Xtest)
tfidfX = tfidf.transform(vX)
predicted = clf.predict(tfidfX)
```

With just:

```
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier()),
])
predicted = pipeline.fit(Xtrain).predict(Xtrain)
# Now evaluate all steps on test set
predicted = pipeline.predict(Xtest)
```

Taken from stackoverflow

Scikit-Learn and sklearn pipeline

Feature Creation and Feature Engineering steps as Scikit-Learn Objects

- **Transformers** - class that have fit and transform method, it transforms data
- Use of scikit-learn base transformers
  - Inherit class and adjust the fit and transform methods
- Click here for an example

Scikit-Learn and sklearn pipeline

Advantages

- Can be tested, versioned, tracked and controlled
- Can build future models on top
- Good software developer practice
- Leverages the power of acknowledged API
- Data scientists familiar with Pipeline use, reduced over-head
- Engineering steps can be packaged and re-used in future ML models

Disadvantages

- Requires team of software developers to build and maintain
- Overhead for software developers to familiarise with code for sklearn API ⇒ difficulties debugging

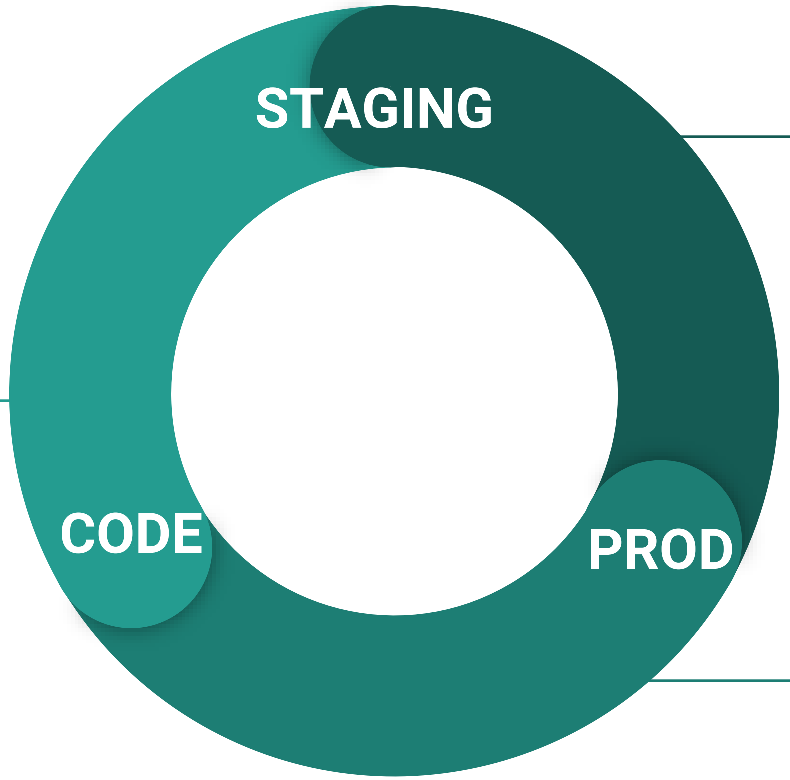
Scikit-Learn and sklearn pipeline: Additional reading resources

- Introduction to Scikit-Learn
- Six reasons why I recommend Scikit-Learn
- Why you should learn Scikit-Learn
- Deep dive into SKlearn pipelines from Kaggle
- SKlearn pipeline tutorial from Kaggle
- Managing Machine Learning workflows with Sklearn pipelines
- A simple example of pipeline in Machine Learning using SKlearn

Should feature selection be part of the Machine Learning automated pipeline?

Feature selection in CI/CD

- New model
- Model refresh



Tests

- Model deployed
- Automatically available to other systems

Feature selection in CI/CD

Advantages

- Reduced overhead in the implementation of the new model
- The new model is almost immediately available to the business systems

Disadvantages

- Lack of data versatility
- No additional data can be fed through the pipeline, as the entire processes are based on the first dataset on which it was built

Feature selection in CI/CD

Including a feature selection algorithm as part of the pipeline,

- Ensures that from all the available features only the most useful ones are selected t train the model
- Potentially avoids overfitting
- Enhances model interpretability

However,

- We would need to deploy code to engineer all available features in the dataset, regardless of whether they will be finally used by the model
- Error handling and unit testing for all the code to engineering features

Feature selection in CI/CD

Suitable:

- Model build and refresh on same data
- Model build and refresh on smaller datasets

Not suitable,

- If model is built using datasets with a high feature space
- If model is constantly enriched with new data sources

Feature selection in the scikit-learn pipe

*C:\Users\Sole\Documents\Udemy\Deployment_MLM\ud-draft\packages\regression_model\regression_model\pipeline.py - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

test_predict.py | test_categorical_encoder.py | pipeline_Example.py | keras.json | pipeline.py

```python
from sklearn.linear_model import Lasso
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

from regression_model.config import config
from regression_model.processing import preprocessors as pp

price_pipe = Pipeline([
                    ('categorical_imputer', pp.CategoricalImputer(variables = config.CATEGORICAL_VARS_WITH_NA)),
                    ('numerical_inputer', pp.NumericalImputer(variables = config.NUMERICAL_VARS_WITH_NA)),
                    ('temporal_variable', pp.TemporalVariableEstimator(variables=config.TEMPORAL_VARS, reference_variable=config.REFERENCE_TEMP_VAR)),
                    ('rare_label_encoder', pp.RareLabelCategoricalEncoder(tol = 0.01, variables = config.CATEGORICAL_VARS)),
                    ('categorical_encoder', pp.CategoricalEncoder(variables=config.CATEGORICAL_VARS)),
                    ('log_transformer', pp.LogTransformer(variables = config.NUMERICALS_LOG_VARS)),
                    ('drop_features', pp.DropUnecessaryFeatures(variables_to_drop = config.DROP_FEATURES)),
                    ('reduce_dim', PCA()),
                    ('scaler', MinMaxScaler()),
                    ('Linear_model', Lasso(alpha=0.005, random_state=0))
                ])
```

Feature selection transformer skeleton

```
import numpy as np
import pandas as pd

from sklearn.base import BaseEstimator, TransformerMixin

from regression_model.processing import errors


# categorical missing value imputer
class MySpecificSelector(BaseEstimator, TransformerMixin):

    def __init__(self, some_param=None):
        self.some_param = some_param

    def fit(self, X, y=None):
        '''Code to select features. Any of your liking'''
        self.selected_features = output of the above code, the selected fetures

        return self

    def transform(self, X):
        X = X.copy()
        X = X[self.selected_features]
        return X
```

How to become a better python developer?

Reading resources to improve coding skills

- The best of the best (coding) practices for python
- Follow links in this StackOverflow thread
- Python best practices for more pythonic code
- The Hitchhiker's guide to python
- Tutorials for pycharm here and here