DSC Capstone Sequence

Lecture 03

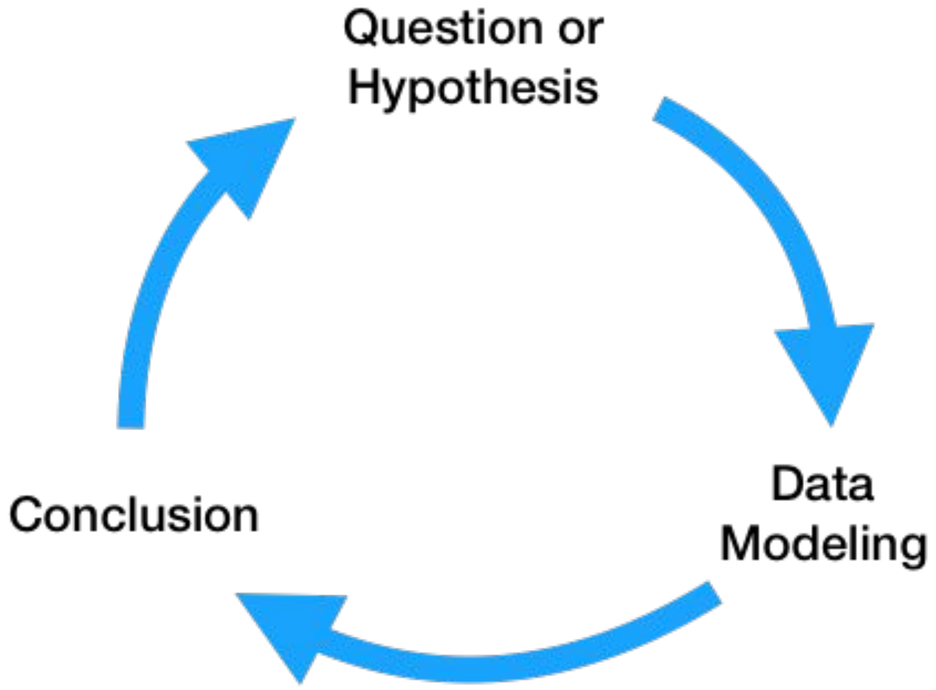
Project Organization

Lecture Outline

- Goals of Data Science Software Development
- Anatomy of a Data Science Project
- Structuring a Data Science Project

The Goals of Data Science Software Development

Data Science Lifecycle

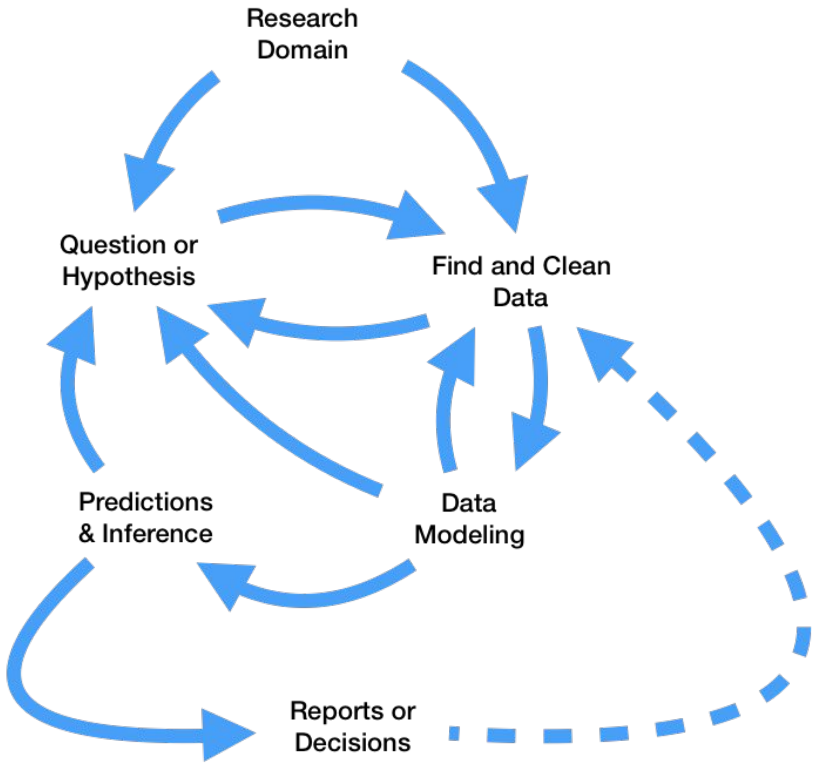


The code for an investigation must:

- Be flexibly written
- Clearly documented.
- Accessible to/Extensible for others.

In order to adapt to successive iterations through the lifecycle.

The *Real* Data Science Lifecycle



Poorly developed code results in:

- Fewer iterations and slower progress on the project
- Higher likelihood of mistakes in the results
- Difficult to understand conclusions
- The project fading into obscurity...

Better code ⇒ higher chance of success

Tools/Libraries for Managing Project Components

Managing Project Components

Too many tools to learn! (and keep up with new developments!)

Instead, learn the core issues:

- What contract does one component need to speak to another?
- Maximize the isolation of each component to enable easy code changes.
- Know each component's relationship to the computational graph:
  - When to recompute a step…
  - When can steps run in parallel?
- How different components scale as the project/data grows in scope?
- Best use of ‘configuration files’ to manage and track iterations.

The Anatomy of a Data Science Project

How each lifecycle component interacts with code

Domain Research

Domain Research informs the bulk of a project’s structure:

- Why you made certain design decisions
- The context behind the quantities of interest
- The subset and kind of data used
- The cleaning logic and any simplifications in modeling

Understanding these choices requires:

- Extensive narrative **documentation** (Notebooks; markdown; pdf)
- **Code comments** to explain specific instances requiring context

Question / Hypothesis

The question being investigated changes as a project evolves.

When the questions are similar:

- Write code parameterized to handle *all* questions simultaneously.
- Choice of parameters ⇔ a different question.
- Parameters are kept in configuration files (e.g. json, ini, cfg, yaml).
  - E.g. Configuration file is an instruction to run 10 instances of the investigation, for 10 different questions, simultaneously on 10 different servers (e.g. questions by year for 2010-2020)
- Strive to write (and rewrite) code to parameterize many possible questions!

Data ETL (extract-transform-load)

As a project evolves, the data may change or new data may be added…

- Keep schema and parameters in configuration (beware: magic numbers!)
- Is the source stable? (DB, API, Scrape)
  - Separate the data ingestion from any transformations
- Unnecessary computation wastes time and resources:
  - Problem: don't re-pull data because your cleaning code changed!
  - Answer: write intermediate files to disk (or a personal file store)
- Write processing code that is agnostic to the computer running it:
  - 'git clone => run' on your laptop or DataHub; scale up only when needed
  - Even better, is the intermediate data accessible from both? (and when do you want that?)

Model Building

Choosing the best model involves exploring *many* parameters!

- Keep track of parameters and results in configuration files.
- Use frameworks that enable 'pipelining' (e.g. sklearn, spark, tensorflow)
- Often need to scale-up processing on different servers

Continued Prediction / Inference

Once a model is built, a project often still lives on…

- Is the finished model being used for live predictions?
    - How does a scikit-learn model get called by a Java backend website?
    - `mdl.predict` may be called via HTTP-requests (RESTful interface).
- Model Quality Reporting:
    - If inference: is the project easily rerun on a new dataset? Can it be automated?
    - For live predictions: are the distributions of predictions stable? What is their quality? Can you create automated reporting?
- What if someone else uses the model? Does it work?
    - Package as a python module or in a Docker Container.

Conclusions / Decision / Report

Once a model is built to your satisfaction...

- Document and explain your results (e.g. in markdown).
- Justify any decisions made from the model
- Create reporting from the model
    - Update the reporting from new data, by rerunning project from scratch.
    - Email the compiled reporting (markdown=>HTML) automatically from a server.

Your project will fade into GitHub obscurity without good documentation!

Summary: Data Science Software Dev

A project should be:

- flexible for quick iterations
  - configuration vs code
- understandable to consumers/users of the output
  - thorough documentation and reports
- usable for developers/researchers extending your work
  - code/api documentation and deployability

Caveat: What if my project isn’t a data analysis?

Some projects may focus on other aspects of data science:

- Developing a method that works with data
- Developing a library for modeling / working with data
- Developing code for data collection

For these projects, you will:

- Follow best-practices for software development in the area
    - Also discussed in this course
- *Still* have a data analysis portion, if only to:
    - Demonstrate the value of your method/library/product
    - Demonstrate usage of whatever you've developed

Caveat: What if my project isn't a data analysis?

In these cases, you will have two repositories:

- The software package that you develop
- The data analysis repository that:
  - demonstrates the value of your work (e.g. in a paper)
  - demonstrates how to use your code (e.g. in documentation)

In these cases, your software repository will be larger and the data analysis repository will be more modest.

This will be discussed more in two weeks!

Structuring a Data Science Project

Configuration vs. Code

Configuration vs Code

Isolate *code that does stuff* from the *parameters the code uses*.

- Code that is used by (multiple other) processes is called *library code*.
- *Configuration* consists of parameters that are passed into library code, and likely to change.
- When to abstract a piece of code to configuration depends on problem
    - Generalize, but don't over-generalize!

Where to place these files?

- Library code in py files; used by functions in scripts or notebooks
- Parameters as variables (imported by script or notebook)

Configuration vs Code

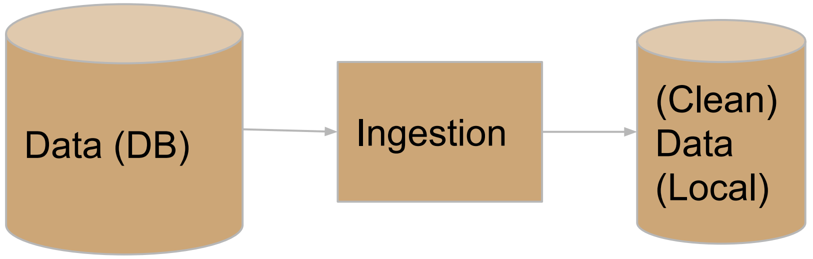
- Simple project separates:
  - library code, configuration
  - code that produces results/analysis (scripts / notebooks)
- More complicated projects have *directories* separating each of these!

```
Project
    |
    |----- code.py
    |----- config.txt
    |----- script.py
    |----- explore.ipynb
```

```
# Inside explore.ipynb / script.py
...
import code
params =
json.load(open('config.txt'))

code.run_process(**params)
```

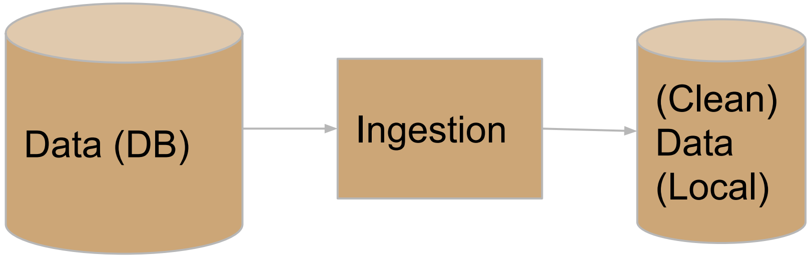
Example: Data Ingestion

- Task: get data from internet to computer.
- Make it easy to (incrementally) change data ingested.
- Rerun to ‘refresh’ with new data.
- Run data pull on different servers to reproduce results



Data Ingestion: `etl.py`

- *Library code*: functions for use by other processes (nb, py files).
  - Good for interactive use; reusable.
  - Written as generically as practicable.
- Contains logic not necessary for a *consumer* of the project to know.
- Contains data collection logic other *developers* might want to expand on, when forking a project.
- Library code know nothing of what calls it!



```
Project
├── README.md
├── data-params.json
├── etl.py
└── run.py
```

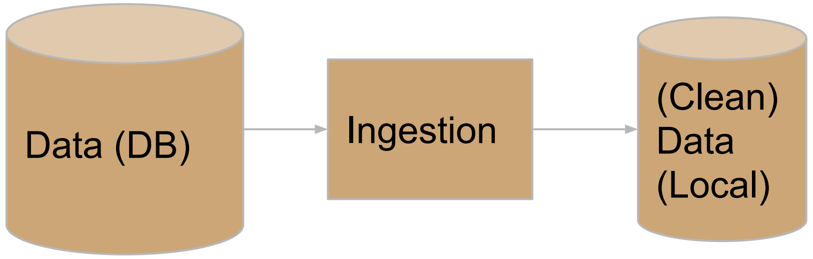
Data Ingestion: `etl.py`

- *Library code*: functions for use by other processes (nb, py files).
    - Good for interactive use; reusable.
    - Written as generically as practicable.
- Contains logic not necessary for a *consumer* of the project to know.
- Contains data collection logic other *developers* might want to expand on, when forking a project.
- Library code know nothing of what calls it!

```
'''
etl.py contains functions used to download tables for different
teams and years.
'''

def get_season(team, year):
    '''
    return a table of season statistics for a
    given team and year.
    '''
    ...
    return ...


def get_data(years, teams, outdir):
    '''
    downloads and saves tables at the specified output
directory
    for the given years and teams.

    :param: years: a list of seasons to collect
    :param: teams: a list of teams to collect
    :param: outpath: the directory to which to save the data.
    '''
    ...
    return
```

Data Ingestion: `data-params.json`

- Configuration: parameters for different investigations and experiments.
- E.g. Parameterize across time/space.
- Used by the *consumer* of the project. Shouldn't require a knowledge of the source code!
- Helps log the results of different experiments.



```
Project
├── README.md
├── data-params.json
├── etl.py
└── run.py
```

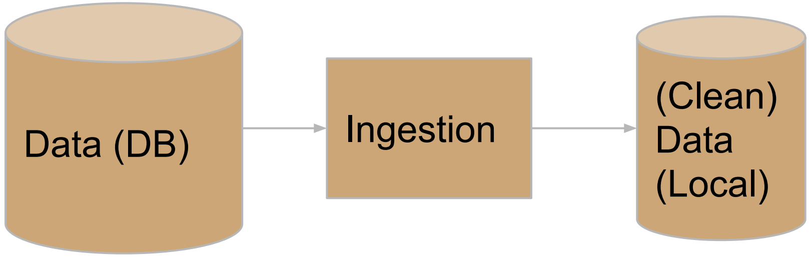
Data Ingestion: `data-params.json`

- Configuration: parameters for different investigations and experiments.
- E.g. Parameterize across time/space.
- Used by the *consumer* of the project. Shouldn't require a knowledge of the source code!
- Helps log the results of different experiments.

```json
{
    "years": [2015, 2016, 2017, 2018, 2019],
    "teams": ["sfo", "gnb"],
    "outpath": "data/raw"
}
```

Data Ingestion: `run.py`

- Script: Builds (common portions of) the project.
- Imports and runs library code: gives examples of *code usage*.
- Current: hand-made `run.py`
- Also possible to use specialized tools: python CLI (e.g. argparse), Makefiles, Maven, etc...



```
Project
├── README.md
├── data-params.json
├── etl.py
└── run.py
```

Data Ingestion: `run.py`

- Imports library code (`get_data`)
- Run as a script:
  - `python run.py data`
- Shebang: `#!/usr/bin/env python`
  - Specifies which python interpreter to use.
- `main` function strings together library functions with parameters in config.
- `__name__ == '__main__'...` returns true only when file is run as a script. Should only have minimal code inside.

```python
#!/usr/bin/env python

import sys
import json

from etl import get_data


def main(targets):

    if 'data' in targets:
        with open('data-params.json') as fh:
            data_cfg = json.load(fh)

        # make the data target
        get_data(**data_cfg)

    return


if __name__ == '__main__':
    targets = sys.argv[1:]
    main(targets)
```

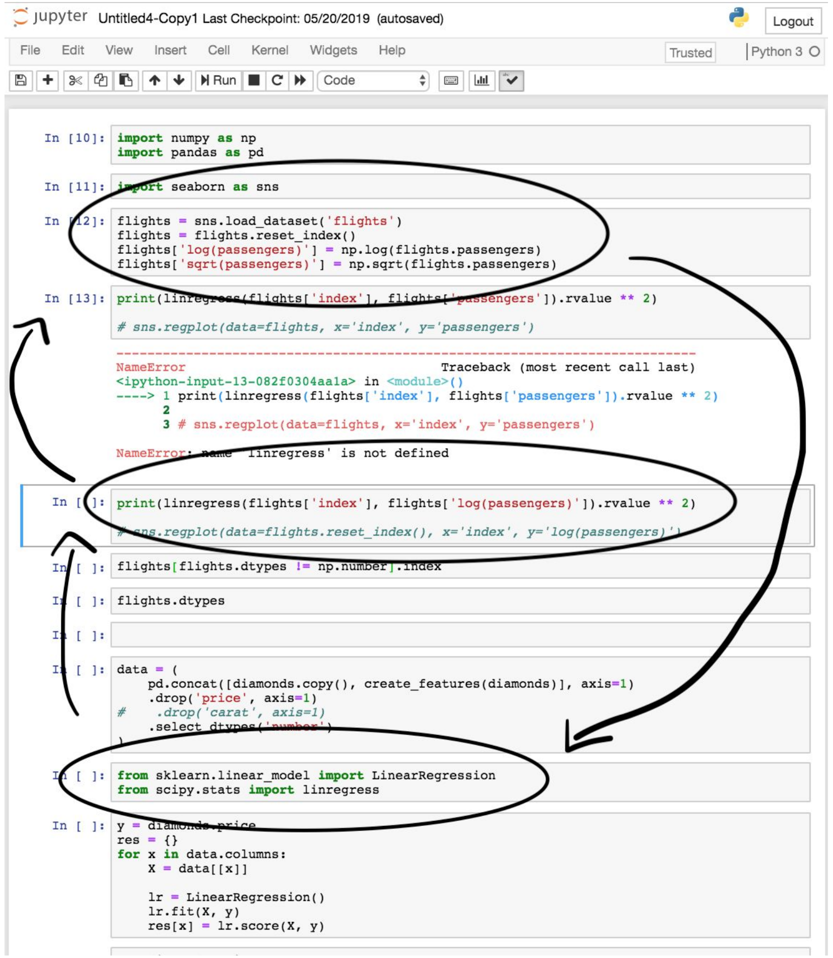
Structuring a Data Science Project

Templates and Cookie Cutter Data Science

Project Structure:

- As project grows, so does code complexity!
- As a project grows, it becomes unclear:
  - how code should be run...
  - what the code *does*...
  - if the code is correct...
- To simple to be realistic:

```
Project
├── README.md
├── data-params.json
├── etl.py
└── run.py
```


jupyter Untitled4-Copy1 Last Checkpoint: 05/20/2019 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [10]: import numpy as np
         import pandas as pd

In [11]: import seaborn as sns

In [12]: flights = sns.load_dataset('flights')
         flights = flights.reset_index()
         flights['log(passengers)'] = np.log(flights.passengers)
         flights['sqrt(passengers)'] = np.sqrt(flights.passengers)

In [13]: print(linregress(flights['index'], flights['passengers']).rvalue ** 2)

         # sns.regplot(data=flights, x='index', y='passengers')

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-082f0304aa1a> in <module>()
----> 1 print(linregress(flights['index'], flights['passengers']).rvalue ** 2)
      2
      3 # sns.regplot(data=flights, x='index', y='passengers')

NameError: name 'linregress' is not defined

In [ ]: print(linregress(flights['index'], flights['log(passengers)']).rvalue ** 2)

        # sns.regplot(data=flights.reset_index(), x='index', y='log(passengers)')

In [ ]: flights[flights.dtypes != np.number].index

In [ ]: flights.dtypes

In [ ]:

In [ ]: data = (
            pd.concat([diamonds.copy(), create_features(diamonds)], axis=1)
            .drop('price', axis=1)
        #     .drop('carat', axis=1)
            .select_dtypes('number')
        )

In [ ]: from sklearn.linear_model import LinearRegression
        from scipy.stats import linregress

In [ ]: y = diamonds.price
        res = {}
        for x in data.columns:
            X = data[[x]]

            lr = LinearRegression()
            lr.fit(X, y)
            res[x] = lr.score(X, y)
```

Why Care About Project Structure?

We're not talking about bikeshedding the indentation aesthetics or pedantic formatting standards — ultimately, data science code quality is about correctness and reproducibility.

Cookie Cutter Data Science

- Clear and consistent project organization encourages software development best-practices and readable code.
- Such habits yield more consistently correct code that’s more easily fixed and adapted to other tasks.
- We will follow the opinions of Cookie Cutter Data Science

An *Example* Project Template

```
├── .gitignore          <- files to keep out of version control (e.g. data/binaries)
├── run.py              <- run.py with commands like `make data` or `make train`
├── README.md           <- The top-level README for developers using this project.
├── data
│   ├── temp            <- Intermediate data that has been transformed.
│   ├── out             <- The final, canonical data sets for modeling.
│   └── raw             <- The original, immutable data dump.
├── notebooks           <- Jupyter notebooks (presentation only).
├── references          <- Data dictionaries, explanatory materials.
├── requirements.txt    <- For reproducing the analysis environment, e.g.
│                          generated with `pip freeze > requirements.txt`
├── src                 <- Source code for use in this project.
    ├── data            <- Scripts to download or generate data
    │   └── make_dataset.py
    ├── features        <- Scripts to turn raw data into features for modeling
    │   └── build_features.py
    ├── models          <- Scripts to train models and make predictions
    │   ├── predict_model.py
    │   └── train_model.py
    └── visualization   <- Scripts to create exploratory and results oriented viz
        └── visualize.py
```

Results are Derived from Immutable Raw Data

- Data is immutable: *never* edit raw data
    - Raw data is always (re)ingested from elsewhere.
    - File-path may be a *symbolic link* (shortcut), if stored locally.
    - **Raw data never changes => doesn’t need version control! (.gitignore)**
- Final data is always reproducible from raw data (with run.py)
- Temp data holds data ‘useful to keep around’ for development, analysis, debugging, etc...

```
...
├── data
│   ├── temp            <- Intermediate data that has been transformed.
│   ├── out             <- The final, canonical data sets for modeling.
│   └── raw             <- The original, immutable data dump.
...
```

Notebooks are for Analysis and Communication

- Notebooks are great for communication, analysis, and initial development.
  - Use to create up-to-date, reproducible, static HTML reports.
- Complicated code in notebooks are hard to understand and don't work well with version control and collaboration.
- Notebooks should:
  - Mostly call library functions in src, with very simple code logic.
  - Never "copy-paste" code between notebooks -- if it's reusable, put it in a library function.
  - Name it something descriptive: `03-fraenkel-prelim-EDA.ipynb`

```
...
├── notebooks          <- Jupyter notebooks (presentation only).
...
```

Build from the Environment Up

- To reproduce a project from scratch, must also reproduce the computational environment on which it was run.
- requirements.txt contains all python libraries needed for running the project.
- `git clone project => mk virtualenv => pip install requirements.txt`
- When a project has more complicated requirements, may need to use container approach (e.g. Docker or Vagrant).

```
...
├── requirements.txt   <- For reproducing the analysis environment, e.g.
│                         generated with `pip freeze > requirements.txt`
...
```