**Requirements**

It's important to make sure anyone can install and run your work. Even if your work is a Jupyter Notebook, there may be packages other people need to install. You can list out all the packages your project is using with pip freeze if you're using a virtual environment. You'll get output like this:

A black rectangular object with white border

AI-generated content may be incorrect.

You'll want to copy these requirements into a folder in your project called requirements.txt.

Now, anyone can install the requirements for your project using:

$ pip install -r requirements.txt

**Paths**

It's common when you're working locally to hardcode absolute paths to data files, like /Users/vik/Documents/data.csv. Other people who want to run your project won't have those same paths on their computer, so they won't be able to run your project. You can fairly easily replace these with relative paths, which allow people who have the data in the same folder as your project, but don't have the same absolute paths, to use your code. Let's say we have this code:

with open("/Users/vik/Documents/data.csv") as f:

data = f.read()

Let's say our project code is at /Users/vik/Documents/Project.ipynb. We can replace the code with a relative path, like:

with open("data.csv") as f:

data = f.read()

It's generally a good idea to put the data in the same folder as your project, or in a subfolder, to make relative paths and loading the data easier.

**Additional files**

By default, running git add . and git commit -m "Message"" will add all the files in a folder to a git commit. However, there are many artifact files that you don't want added. Here's an example folder:

loans

│ \_\_pycache\_\_

│ main.py

│ main.pyc

│ temp.json

│

└───data

│ test.csv

│ train.csv

Note files like \_\_pycache\_\_, main.pyc, and temp.json. The main code of the project is in main.py, and the data is in data/test.csv, and data/train.csv. For someone to run the project, those are the only files they need. Folders like \_\_pycache\_\_ and main.pyc are automatically generated by Python when we run code or install packages. These enable Python scripts and package installation to be faster and more reliable. However, these files aren't part of your project, and thus shouldn't be distributed to others. We can ignore files like this with a .gitignore file. We can add a .gitignore file to our folder:

loans

│ .gitignore

│ \_\_pycache\_\_

│ main.py

│ main.pyc

│ temp.json

│

└───data

│ test.csv

│ train.csv

The content of the .gitignore file is a list of files to ignore. We can create a .gitignore file, then add \*.pyc and \_\_pycache\_\_ to ignore the generated files in our folder:

\*.pyc

\_\_pycache\_\_

This still leaves the temp.json file. We can add another line to ignore this file:

\*.pyc

\_\_pycache\_\_

temp.json

This will ensure that these files are not tracked by git, and added to new git commits when you run git add .. However, if you've already added the files to a git commit before, you'll need to remove them first with git rm temp.json --cached.

**Secret keys or files**

Many projects use secret keys to access resources. A good example is api keys, such as AWS\_ACCESS\_KEY="3434ffdsfd". You absolutely don't want to share your secret keys with other people -- this allows them to access your resources, and could cost you money.

First, we create a file called settings.py, with the following lines:

API\_KEY = ""

try:

from .private import \*

except Exception:

pass

The above code defines a key called API\_KEY. It also tries to import from a file called private.py, and doesn't do anything if the file doesn't exist. We then need to add a private.py with the following content:

API\_KEY = "34343434fdfddf"

Then, we need to add private.py to .gitignore so it doesn't get committed:

private.py

Then, we modify our original code:

import settings

forecast = forecastio.load\_forecast(settings.API\_KEY, 37.77493, -122.41942)

All the changes we've made above result in the following:

* The code imports the settings file
* The settings file imports the private.py file
  + This overwrites the API\_KEY variable in the settings file with the API\_KEY defined in the private file
* The code uses API\_KEY from the settings file, which equals "34343434fdfddf"

The next time you make a git commit, private.py will be ignored. However, if someone else looks at your repository, they'll see that they need to fill out settings.py with their own settings to get things to work properly. So everything will work for you, you won't share your secret keys with others, and things will work for others.

**The README file**

The README file is very critical to your project. The README is usually named README.md, and is in [Markdown](https://daringfireball.net/projects/markdown/syntax) format. GitHub will automatically parse Markdown format and render it. Your README file should describe:

* The goals of your project
* Your though process and methods in creating the project
* How to install your project
* How to run your project

You can find good README examples [here](https://github.com/VikParuchuri/apartment-finder/blob/master/README.md) and [here](https://github.com/dataquestio/loan-prediction/blob/master/README.md). It's important to go through the installation steps yourself in a new folder or on a new computer, to make sure everything works. The README is also the first and potentially only thing someone will look at, because GitHub renders it below the repository file view. It's important to "sell" what the project is, why you made it, and what's interesting about it.

**Inline explanations**

If you're writing Python script files, you'll want to include lots of inline comments to make your logic easier to follow.

A better alternative is:

def count\_performance\_rows():

"""

A function to count the number of rows that deal with performance for each loan.

Each row in the source text file is a loan\_id and date.

If there's a date, it means the loan was foreclosed on.

We'll return a dictionary that indicates if each loan was foreclosed on, along with the number of performance events per loan.

"""

counts = {}

# Read the data file.

with open(os.path.join(settings.PROCESSED\_DIR, "Performance.txt"), 'r') as f:

for i, line in enumerate(f):

if i == 0:

# Skip the header row

continue

# Each row is a loan id and a date, separated by a |

loan\_id, date = line.split("|")

# Convert to integer

loan\_id = int(loan\_id)

# Add the loan to the counts dictionary, so we can count the number of performance events.

if loan\_id not in counts:

counts[loan\_id] = {

"foreclosure\_status": False,

"performance\_count": 0

}

# Increment the counter.

counts[loan\_id]["performance\_count"] += 1

# If there's a date, it indicates that the loan was foreclosed on

if len(date.strip()) > 0:

counts[loan\_id]["foreclosure\_status"] = True

return counts

* Add your projects to your [LinkedIn](https://www.linkedin.com/) profile
* List your projects on your resume

**Supplementing the data**

Once you have a good topic, it’s good to scope out other datasets that can enhance the topic or give you more depth to explore. It’s good to do this upfront, so you have as much data as possible to explore as you’re building your project. Having too little data might mean that you give up on your project too early.

 A well structured project follows a few principles:

* Separates data files and code files.
* Separates raw data from generated data.
* Has a README.md file that walks people through installing and using the project.
* Has a requirements.txt file that contains all the packages needed to run the project.
* Has a single settings.py file that contains any settings that are used in other files.
  + For example, if you are reading the same file from multiple Python scripts, it's useful to have them all import settings and get the file name from a centralized place.
* Has a .gitignore file that prevents large or secret files from being committed.
* Breaks each step in our task into a separate file that can be executed separately.
  + For example, we may have one file for reading in the data, one for creating features, and one for making predictions.
* Stores intermediate values. For example, one script may output a file that the next script can read.
  + This enables us to make changes in our data processing flow without recalculating everything.

Our file structure will look something like this shortly:

loan-prediction

├── data

├── processed

├── .gitignore

├── README.md

├── requirements.txt

├── settings.py