* **PRODUCTION CODE:** software running on production servers to handle live users and data of the intended audience. Note this is different from *production quality code*, which describes code that meets expectations in reliability, efficiency, etc., for production. Ideally, all code in production meets these expectations, but this is not always the case.
* **CLEAN:** readable, simple, and concise. A characteristic of production quality code that is crucial for collaboration and maintainability in software development.
* **MODULAR:** logically broken up into functions and modules. Also an important characteristic of production quality code that makes your code more organized, efficient, and reusable.
* **MODULE:** a file. Modules allow code to be reused by encapsulating them into files that can be imported into other files.

**Documentation**

* **DOCUMENTATION:** additional text or illustrated information that comes with or is embedded in the code of software.
* Helpful for clarifying complex parts of code, making your code easier to navigate, and quickly conveying how and why different components of your program are used.
* Several types of documentation can be added at different levels of your program:
  + **In-line Comments** - line level
  + **Docstrings** - module and function level
  + **Project Documentation** - project level

# Docstrings

Docstring, or documentation strings, are valuable pieces of documentation that explain the functionality of any function or module in your code. Ideally, each of your functions should always have a docstring.

Docstrings are surrounded by triple quotes. The first line of the docstring is a brief explanation of the function's purpose.

### One line docstring

**def** **population\_density**(population, land\_area):

"""Calculate the population density of an area."""

**return** population / land\_area

If you think that the function is complicated enough to warrant a longer description, you can add a more thorough paragraph after the one line summary.

### Multi line docstring

**def** **population\_density**(population, land\_area):

"""Calculate the population density of an area.

Args:

population: int. The population of the area

land\_area: int or float. This function is unit-agnostic, if you pass in values in terms of square km or square miles the function will return a density in those units.

Returns:

population\_density: population/land\_area. The population density of a

particular area.

"""

**return** population / land\_area

The next element of a docstring is an explanation of the function's arguments. Here you list the arguments, state their purpose, and state what types the arguments should be. Finally it is common to provide some description of the output of the function. Every piece of the docstring is optional; however, doc strings are a part of good coding practice.

### Resources:

* [**PEP 257 - Docstring Conventions**](https://www.python.org/dev/peps/pep-0257/)
* [**NumPy Docstring Guide**](https://numpydoc.readthedocs.io/en/latest/format.html)

**Project Documentation**

Project documentation is essential for getting others to understand why and how your code is relevant to them, whether they are potentials users of your project or developers who may contribute to your code. A great first step in project documentation is your README file. It will often be the first interaction most users will have with your project.

Whether it's an application or a package, your project should absolutely come with a README file. At a minimum, this should explain what it does, list its dependencies, and provide sufficiently detailed instructions on how to use it. You want to make it as simple as possible for others to understand the purpose of your project, and quickly get something working.

Translating all your ideas and thoughts formally on paper can be a little difficult, but you'll get better over time and makes a significant difference in helping others realize the value of your project. Writing this documentation can also help you improve the design of your code, as you're forced to think through your design decisions more thoroughly. This also allows future contributors to know how to follow your original intentions.

A full Udacity course on this topic can be found [**here**](https://classroom.udacity.com/courses/ud777).

Here are a few READMEs from some popular projects:

* [**Bootstrap**](https://github.com/twbs/bootstrap)
* [**Scikit-learn**](https://github.com/scikit-learn/scikit-learn)
* [**Stack Overflow Blog**](https://github.com/jjrunner/stackoverflow)

# Scenario #1

Let's walk through the git commands that go along with each step in the scenario you just observed in the video above.

#### STEP 1: You have a local version of this repository on your laptop, and to get the latest stable version, you pull from the develop branch.

##### Switch to the develop branch

git checkout develop

##### Pull latest changes in the develop branch

git pull

#### STEP 2: When you start working on this demographic feature, you create a new branch for this called demographic, and start working on your code in this branch.

##### Create and switch to new branch called demographic from develop branch

git checkout -b demographic

##### Work on this new feature and commit as you go

git commit -m 'added gender recommendations'  
git commit -m 'added location specific recommendations'  
...

#### STEP 3: However, in the middle of your work, you need to work on another feature. So you commit your changes on this demographic branch, and switch back to the develop branch.

##### Commit changes before switching

git commit -m 'refactored demographic gender and location recommendations '

##### Switch to the develop branch

git checkout develop

#### STEP 4: From this stable develop branch, you create another branch for a new feature called friend\_groups.

##### Create and switch to new branch called friend\_groups from develop branch

git checkout -b friend\_groups

#### STEP 5: After you finish your work on the friend\_groups branch, you commit your changes, switch back to the development branch, merge it back to the develop branch, and push this to the remote repository’s develop branch.

##### Commit changes before switching

git commit -m 'finalized friend\_groups recommendations '

##### Switch to the develop branch

git checkout develop

##### Merge friend\_groups branch to develop

git merge --no-ff friends\_groups

##### Push to remote repository

git push origin develop

#### STEP 6: Now, you can switch back to the demographic branch to continue your progress on that feature.

##### Switch to the demographic branch

git checkout demographic

# Scenario #2

Let's walk through the git commands that go along with each step in the scenario you just observed in the video above.

#### Step 1: You check your commit history, seeing messages of the changes you made and how well it performed.

##### View log history

git log

#### Step 2: The model at this commit seemed to score the highest, so you decide to take a look.

##### Checkout a commit

git checkout bc90f2cbc9dc4e802b46e7a153aa106dc9a88560

After inspecting your code, you realize what modifications made this perform well, and use those for your model.

#### Step 3: Now, you’re pretty confident merging this back into the development branch, and pushing the updated recommendation engine.

##### Switch to develop branch

git checkout develop

##### Merge friend\_groups branch to develop

git merge --no-ff friend\_groups

##### Push changes to remote repository

git push origin develop

# Scenario #3

Let's walk through the git commands that go along with each step in the scenario you just observed in the video above.

#### Step 1: Andrew commits his changes to the documentation branch, switches to the development branch, and pulls down the latest changes from the cloud on this development branch, including the change I merged previously for the friends group feature.

##### Commit changes on documentation branch

git commit -m "standardized all docstrings in process.py"

##### Switch to develop branch

git checkout develop

##### Pull latest changes on develop down

git pull

#### Step 2: Then, Andrew merges his documentation branch on the develop branch on his local repository, and then pushes his changes up to update the develop branch on the remote repository.

##### Merge documentation branch to develop

git merge --no-ff documentation

##### Push changes up to remote repository

git push origin develop

#### Step 3: After the team reviewed both of your work, they merge the updates from the development branch to the master branch. Now they push the changes to the master branch on the remote repository. These changes are now in production.

##### Merge develop to master

git merge --no-ff develop

##### Push changes up to remote repository

git push origin master

### Resources

There's a great article on a successful git branching strategy that you should really read [**here**](http://nvie.com/posts/a-successful-git-branching-model/).

### Note on Merge Conflicts

For the most part, git makes merging changes between branches really simple. However, there are some cases where git will be confused on how to combine two changes, and asks you for help. This is called a merge conflict.

Mostly commonly, this happens when two branches modify the same file.

For example, in this situation, let’s say I deleted a line that Andrew modified on his branch. Git wouldn’t know whether to delete the line or modify it. Here, you need to tell git which change to take, and some tools even allow you to edit the change manually. If it isn’t straightforward, you may have to consult with the developer of the other branch to handle a merge conflict.

You can learn more about merge conflicts and methods to handle them [**here**](https://help.github.com/articles/about-merge-conflicts/).

NEXT

**Model Versioning**

In the previous example, you may have noticed that each commit was documented with a score for that model. This is one simple way to help you keep track of model versions. Version control in data science can be tricky, because there are many pieces involved that can be hard to track, such as large amounts of data, model versions, seeds, hyperparameters, etc.

Here are some resources for useful ways and tools for managing versions of models and large data. These are here for you to explore, but are not necessary to know now as you start your journey as a data scientist. On the job, you’ll always be learning new skills, and many of them will be specific to the processes set in your company.

* [**How to Version Control Your Production Machine Learning Models**](https://blog.algorithmia.com/how-to-version-control-your-production-machine-learning-models/)
* [**Versioning Data Science**](https://shuaiw.github.io/2017/07/30/versioning-data-science.html)