**Software Engineering Practices**

**Part 1**

In this lesson, you'll learn about the following practices of software engineering and how they apply in data science.

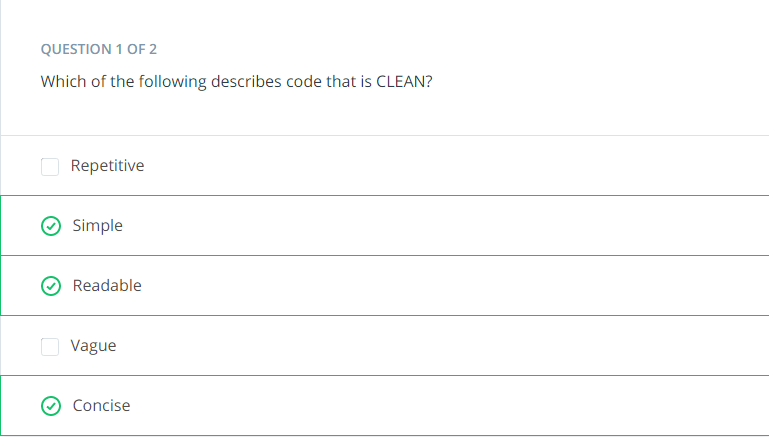
* Writing clean and modular code
* Writing efficient code
* Code refactoring
* Adding meaningful documentation
* Using version control

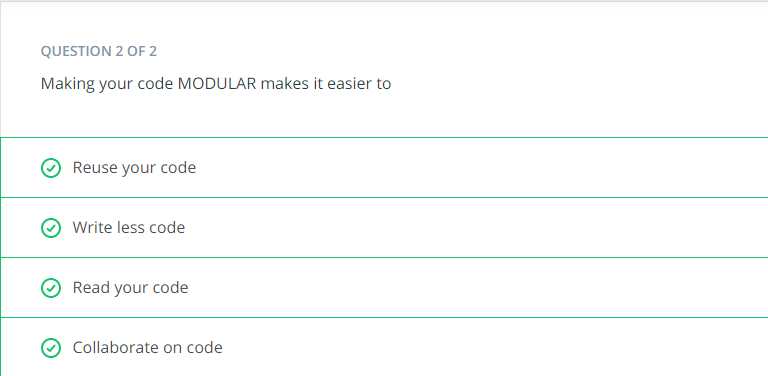
In the lesson following this one (Part II) you'll also learn these software engineering practices:

* Testing
* Logging
* Code reviews

**Clean and Modular Code**

* **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.

****



# Refactoring Code

* **REFACTORING:** Restructuring your code to improve its internal structure, without changing its external functionality. This gives you a chance to clean and modularize your program after you've got it working.
* Since it isn't easy to write your best code while you're still trying to just get it working, allocating time to do this is essential to producing high quality code. Despite the initial time and effort required, this really pays off by speeding up your development time in the long run.
* You become a much stronger programmer when you're constantly looking to improve your code. The more you refactor, the easier it will be to structure and write good code the first time.

# Writing Clean Code

# <https://www.youtube.com/watch?v=wNaiahWCwkQ&feature=youtu.be> (for video)

#### Tip: Use meaningful names

* ***Be descriptive and imply type****-* E.g. for Booleans, you can prefix with is\_ or has\_ to make it clear it is a condition. You can also use part of speech to imply types, like verbs for functions and nouns for variables.
* ***Be consistent but clearly differentiate****-* E.g. age\_list and age is easier to differentiate than ages and age.
* ***Avoid abbreviations and especially single letters****-* (Exception: counters and common math variables) Choosing when these exceptions can be made can be determined based on the audience for your code. If you work with other data scientists, certain variables may be common knowledge. While if you work with full stack engineers, it might be necessary to provide more descriptive names in these cases as well.
* ***Long names != descriptive names****-* You should be descriptive, but only with relevant information. E.g. good functions names describe what they do well without including details about implementation or highly specific uses.

Try testing how effective your names are by asking a fellow programmer to guess the purpose of a function or variable based on its name, without looking at your code. Coming up with meaningful names often requires effort to get right.

#### Tip: Use whitespace properly

* Organize your code with consistent indentation - the standard is to use 4 spaces for each indent. You can make this a default in your text editor.
* Separate sections with blank lines to keep your code well organized and readable.
* *Try to limit your lines to around 79 characters, which is the guideline given in the PEP 8 style guide. In many good text editors, there is a setting to display a subtle line that indicates where the 79 character limit is.(* <https://www.python.org/dev/peps/pep-0008/?#code-lay-out>*)*

# Quiz : Categorizing Tasks

# 

# Quiz: Buying Stocks

Imagine you analyzed several stocks and calculated the ideal price, or limit price, you'd want to buy each stock at. You write a program to iterate through your stocks and buy it if the current price is below or equal to the limit price you computed. Otherwise, you put it on a watchlist. Below are three ways of writing this code. Which of the following is the most clean?

*# Choice A*

stock\_limit\_prices = {'LUX': 62.48, 'AAPL': 127.67, 'NVDA': 161.24}

**for** stock\_ticker, stock\_limit\_price **in** buy\_prices.items():

**if** stock\_limit\_price <= get\_current\_stock\_price(ticker):

buy\_stock(ticker)

**else**:

watchlist\_stock(ticker)

*# Choice B*

prices = {'LUX': 62.48, 'AAPL': 127.67, 'NVDA': 161.24}

**for** ticker, price **in** prices.items():

**if** price <= current\_price(ticker):

buy(ticker)

**else**:

watchlist(ticker)

*# Choice C*

limit\_prices = {'LUX': 62.48, 'AAPL': 127.67, 'NVDA': 161.24}

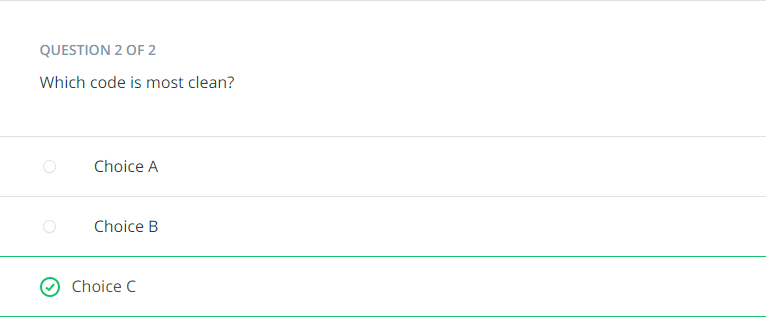
**for** ticker, limit **in** limit\_prices.items():

**if** limit <= get\_current\_price(ticker):

buy(ticker)

**else**:

watchlist(ticker)



# Writing Modular Code

# <https://www.youtube.com/watch?v=qN6EOyNlSnk&feature=youtu.be>

#### Tip: DRY (Don't Repeat Yourself)

Don't repeat yourself! Modularization allows you to reuse parts of your code. Generalize and consolidate repeated code in functions or loops.

#### Tip: Abstract out logic to improve readability

*Abstracting out code into a function not only makes it less repetitive, but also improves readability with descriptive function names. Although your code can become more readable when you abstract out logic into functions, it is possible to over-engineer this and have way too many modules, so use your judgement.*

#### Tip: Minimize the number of entities (functions, classes, modules, etc.)

There are trade-offs to having function calls instead of inline logic. If you have broken up your code into an unnecessary amount of functions and modules, you'll have to jump around everywhere if you want to view the implementation details for something that may be too small to be worth it. Creating more modules doesn't necessarily result in effective modularization.

#### Tip: Functions should do one thing

Each function you write should be focused on doing one thing. If a function is doing multiple things, it becomes more difficult to generalize and reuse. Generally, if there's an "and" in your function name, consider refactoring.

#### Tip: Arbitrary variable names can be more effective in certain functions

Arbitrary variable names in general functions can actually make the code more readable.

#### Tip: Try to use fewer than three arguments per function

Try to use no more than three arguments when possible. This is not a hard rule and there are times it is more appropriate to use many parameters. But in many cases, it's more effective to use fewer arguments. Remember we are modularizing to simplify our code and make it more efficient to work with. If your function has a lot of parameters, you may want to rethink how you are splitting this up.

**Efficient Code**

Knowing how to write code that runs efficiently is another essential skill in software development. Optimizing code to be more efficient can mean making it:

* Execute faster
* Take up less space in memory/storage

The project you're working on would determine which of these is more important to optimize for your company or product. When we are performing lots of different transformations on large amounts of data, this can make orders of magnitudes of difference in performance.

# Optimizing - Common Books

# <https://www.youtube.com/watch?v=WF9n_19V08g&feature=youtu.be>

### Resources:

* [**What makes sets faster than lists**](https://stackoverflow.com/questions/8929284/what-makes-sets-faster-than-lists-in-python/8929445)

# 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

**In-Line Comments**

[**https://www.youtube.com/watch?v=-G6yg3Xhl8I&feature=youtu.be**](https://www.youtube.com/watch?v=-G6yg3Xhl8I&feature=youtu.be)

* In-line comments are text following hash symbols throughout your code. They are used to explain parts of your code, and really help future contributors understand your work.
* One way comments are used is to document the major steps of complex code to help readers follow. Then, you may not have to understand the code to follow what it does. However, others would argue that this is using comments to justify bad code, and that if code requires comments to follow, it is a sign refactoring is needed.
* Comments are valuable for explaining where code cannot. For example, the history behind why a certain method was implemented a specific way. Sometimes an unconventional or seemingly arbitrary approach may be applied because of some obscure external variable causing side effects. These things are difficult to explain with code.

# Docstrings

[**https://www.youtube.com/watch?v=\_gapemxsRJY&feature=youtu.be**](https://www.youtube.com/watch?v=_gapemxsRJY&feature=youtu.be)

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. The two types are :

### One line docstring

### Multi line docstring

### 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)

# Quiz: Documentation

# 

# 

# Version Control in Data Science

For videos go to directory.

# 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/).

**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)