**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: Meaningful Names

#### 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.

# Writing Clean Code: Nice Whitespace

#### 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.*

For more guidelines, check out the code layout section of PEP 8 in the notes below.

# Writing Clean Code: Nice Whitespace

[PEP 8 guidelines for code layout](https://www.python.org/dev/peps/pep-0008/?#code-lay-out)

# Writing Modular Code

#### 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 tradeoffs 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.

**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

**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/).

In part 2 of software engineering practices, you'll learn about the following practices of software engineering and how they apply in data science.

* Testing
* Logging
* Code reviews

# Testing

Testing your code is essential before deployment. It helps you catch errors and faulty conclusions before they make any major impact. Today, employers are looking for data scientists with the skills to properly prepare their code for an industry setting, which includes testing their code.

# Testing And Data Science

* Problems that could occur in data science aren’t always easily detectable; you might have values being encoded incorrectly, features being used inappropriately, unexpected data breaking assumptions
* To catch these errors, you have to check for the quality and accuracy of your analysis in addition to the quality of your code. Proper testing is necessary to avoid unexpected surprises and have confidence in your results.
* **TEST DRIVEN DEVELOPMENT:** a development process where you write tests for tasks before you even write the code to implement those tasks.
* **UNIT TEST:** a type of test that covers a “unit” of code, usually a single function, independently from the rest of the program.

### Resources:

* Four Ways Data Science Goes Wrong and How Test Driven Data Analysis Can Help: [**Blog Post**](https://www.predictiveanalyticsworld.com/patimes/four-ways-data-science-goes-wrong-and-how-test-driven-data-analysis-can-help/6947/)
* Ned Batchelder: Getting Started Testing: [**Slide Deck**](https://speakerdeck.com/pycon2014/getting-started-testing-by-ned-batchelder) and [**Presentation Video**](https://www.youtube.com/watch?v=FxSsnHeWQBY)

# Unit Tests

We want to test our functions in a way that is repeatable and automated. Ideally, we'd run a test program that runs all our unit tests and cleanly lets us know which ones failed and which ones succeeded. Fortunately, there are great tools available in Python that we can use to create effective unit tests!

#### Unit Test Advantages and Disadvantages

The advantage of unit tests is that they are isolated from the rest of your program, and thus, no dependencies are involved. They don't require access to databases, APIs, or other external sources of information. However, passing unit tests isn’t always enough to prove that our program is working successfully. To show that all the parts of our program work with each other properly, communicating and transferring data between them correctly, we use integration tests. In this lesson, we'll focus on unit tests; however, when you start building larger programs, you will want to use integration tests as well.

You can read about integration testing and how integration tests relate to unit tests [**here**](https://www.fullstackpython.com/integration-testing.html). That article contains other very useful links as well.

**Unit Testing Tools**

To install pytest, run pip install -U pytest in your terminal. You can see more information on getting started [**here**](https://docs.pytest.org/en/latest/getting-started.html).

* Create a test file starting with test\_
* Define unit test functions that start with test\_ inside the test file
* Enter pytest into your terminal in the directory of your test file and it will detect these tests for you!

test\_ is the default - if you wish to change this, you can learn how to in this **[pytest configuration](https://docs.pytest.org/en/latest/customize.html" \t "_blank)**

In the test output, periods represent successful unit tests and F's represent failed unit tests. Since all you see is what test functions failed, it's wise to have only one assert statement per test. Otherwise, you wouldn't know exactly how many tests failed, and which tests failed.

Your tests won't be stopped by failed assert statements, but it will stop if you have syntax errors.

**Test Driven Development and Data Science**

* **TEST DRIVEN DEVELOPMENT:** writing tests before you write the code that’s being tested. Your test would fail at first, and you’ll know you’ve finished implementing a task when this test passes.
* Tests can check for all the different scenarios and edge cases you can think of, before even starting to write your function. This way, when you do start implementing your function, you can run this test to get immediate feedback on whether it works or not in all the ways you can think of, as you tweak your function.
* When refactoring or adding to your code, tests help you rest assured that the rest of your code didn't break while you were making those changes. Tests also helps ensure that your function behavior is repeatable, regardless of external parameters, such as hardware and time.

Test driven development for data science is relatively new and has a lot of experimentation and breakthroughs appearing, which you can learn more about in the resources below.

* [**Data Science TDD**](https://www.linkedin.com/pulse/data-science-test-driven-development-sam-savage/)
* [**TDD for Data Science**](http://engineering.pivotal.io/post/test-driven-development-for-data-science/)
* [**TDD is Essential for Good Data Science Here's Why**](https://medium.com/@karijdempsey/test-driven-development-is-essential-for-good-data-science-heres-why-db7975a03a44)
* [**Testing Your Code**](http://docs.python-guide.org/en/latest/writing/tests/) (general python TDD)

# Logging

Logging is valuable for understanding the events that occur while running your program. For example, if you run your model over night and see that it's producing ridiculous results the next day, log messages can really help you understand more about the context in which this occurred. Lets learn about the qualities that make a log message effective.

# Log Messages

Logging is the process of recording messages to describe events that have occurred while running your software. Let's take a look at a few examples, and learn tips for writing good log messages.

#### Tip: Be professional and clear

Bad: Hmmm... this isn't working???

Bad: idk.... :(

Good: Couldn't parse file.

#### Tip: Be concise and use normal capitalization

Bad: Start Product Recommendation Process

Bad: We have completed the steps necessary and will now proceed with the recommendation process for the records in our product database.

Good: Generating product recommendations.

#### Tip: Choose the appropriate level for logging

DEBUG - level you would use for anything that happens in the program.  
ERROR - level to record any error that occurs  
INFO - level to record all actions that are user-driven or system specific, such as regularly scheduled operations

#### Tip: Provide any useful information

Bad: Failed to read location data

Good: Failed to read location data: store\_id 8324971

Quiz sol:

Great job! ERROR is the appropriate level for this error message, though more information on where, when, how, etc. this occurred would be useful for debugging. If you recall from the last lesson, it's best practice to use concise and clear language that is professional and uses normal capitalization. This way, the message is efficient and easily understandable. The second sentence seems quite unclear and personal, so we should remove that and communicate it elsewhere.

**Code Reviews**

Code reviews benefit everyone in a team to promote best programming practices and prepare code for production. Let's go over what to look for in a code review and some tips on how to conduct one.

* [**Code Review**](https://github.com/lyst/MakingLyst/tree/master/code-reviews)
* [**Code Review Best Practices**](https://www.kevinlondon.com/2015/05/05/code-review-best-practices.html)

# Questions to Ask Yourself When Conducting a Code Review

First, let's look over some of the questions we may ask ourselves while reviewing code. These are simply from the concepts we've covered in these last two lessons!

#### Is the code clean and modular?

* Can I understand the code easily?
* Does it use meaningful names and whitespace?
* Is there duplicated code?
* Can you provide another layer of abstraction?
* Is each function and module necessary?
* Is each function or module too long?

#### Is the code efficient?

* Are there loops or other steps we can vectorize?
* Can we use better data structures to optimize any steps?
* Can we shorten the number of calculations needed for any steps?
* Can we use generators or multiprocessing to optimize any steps?

#### Is documentation effective?

* Are in-line comments concise and meaningful?
* Is there complex code that's missing documentation?
* Do function use effective docstrings?
* Is the necessary project documentation provided?

#### Is the code well tested?

* Does the code high test coverage?
* Do tests check for interesting cases?
* Are the tests readable?
* Can the tests be made more efficient?

#### Is the logging effective?

* Are log messages clear, concise, and professional?
* Do they include all relevant and useful information?
* Do they use the appropriate logging level?