* **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.

# 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.

**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

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

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.

# 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.

# Git

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

### [Resolving merge conflicts](https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-merge-conflicts#resolving-merge-conflicts)

To resolve a merge conflict, you must manually edit the conflicted file to select the changes that you want to keep in the final merge. There are a couple of different ways to resolve a merge conflict:

* If your merge conflict is caused by competing line changes, such as when people make different changes to the same line of the same file on different branches in your Git repository, you can resolve it on GitHub using the conflict editor. For more information, see "[Resolving a merge conflict on GitHub](https://help.github.com/en/articles/resolving-a-merge-conflict-on-github)."
* For all other types of merge conflicts, you must resolve the merge conflict in a local clone of the repository and push the change to your branch on GitHub. You can use the command line or a tool like [GitHub Desktop](https://desktop.github.com/) to push the change. For more information, see "[Resolving a merge conflict on the command line](https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line)."

If you have a merge conflict on the command line, you cannot push your local changes to GitHub until you resolve the merge conflict locally on your computer. If you try merging branches on the command line that have a merge conflict, you'll get an error message. For more information, see "[Resolving a merge conflict using the command line](https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line)."

$ git merge BRANCH-NAME

> Auto-merging styleguide.md

> CONFLICT (content): Merge conflict in styleguide.md

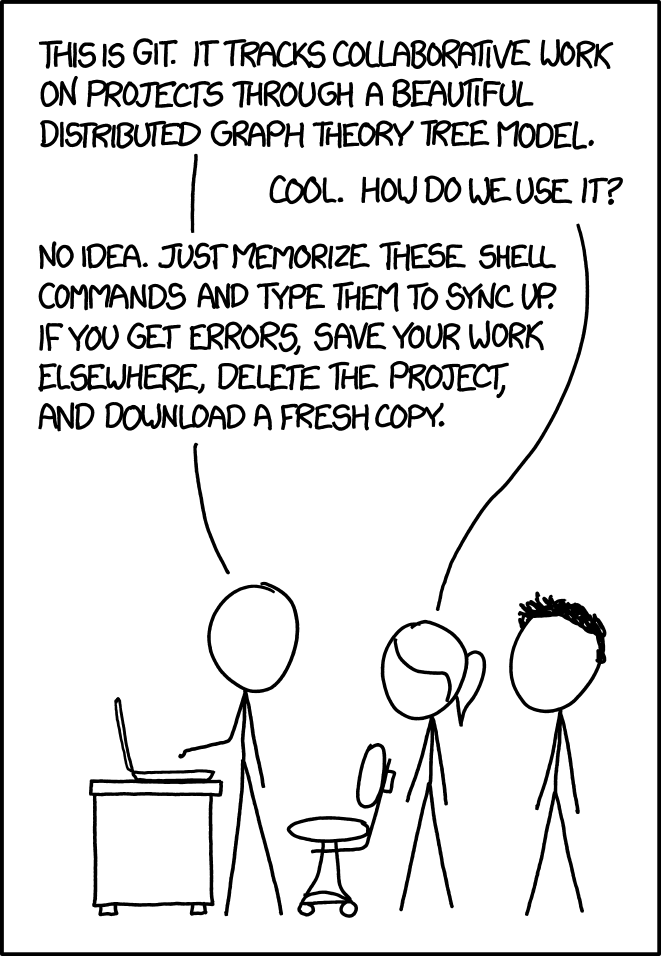
> Automatic merge failed; fix conflicts and then commit the result

## ****Versioning Tools to Get The Job Done****

It’s hard to understate how nascent the field of production Machine Learning is, and that means the tools supporting this ecosystem are only starting to be fully developed. Here are some of the solutions that practitioners are currently using, and some new entrants too.

### 1. Git

Git is the versioning protocol used across the board to monitor and version software development and deployment. You might be familiar with [GitHub](https://github.com/" \t "_blank) or [BitBucket](https://bitbucket.org/" \t "_blank), which are web-based commercial implementations of this open-source tool. Git tracks any changes made to your code and gives you functionality around implementing, storing, and merging those changes. Pretty much everyone uses it in one way or another.



Source: [xkcd](https://xkcd.com/1597/" \t "_blank)

But alas, Git is not without its issues. In addition to the often perplexing nature of using the actual protocol, its missing a lot of the functionality that you need for machine learning (because it wasn’t created for ML!). Git itself doesn’t allow you to track data, changes to model files, and model dependencies. There are extensions that can help, but those solutions are tough to implement and rarely complete.

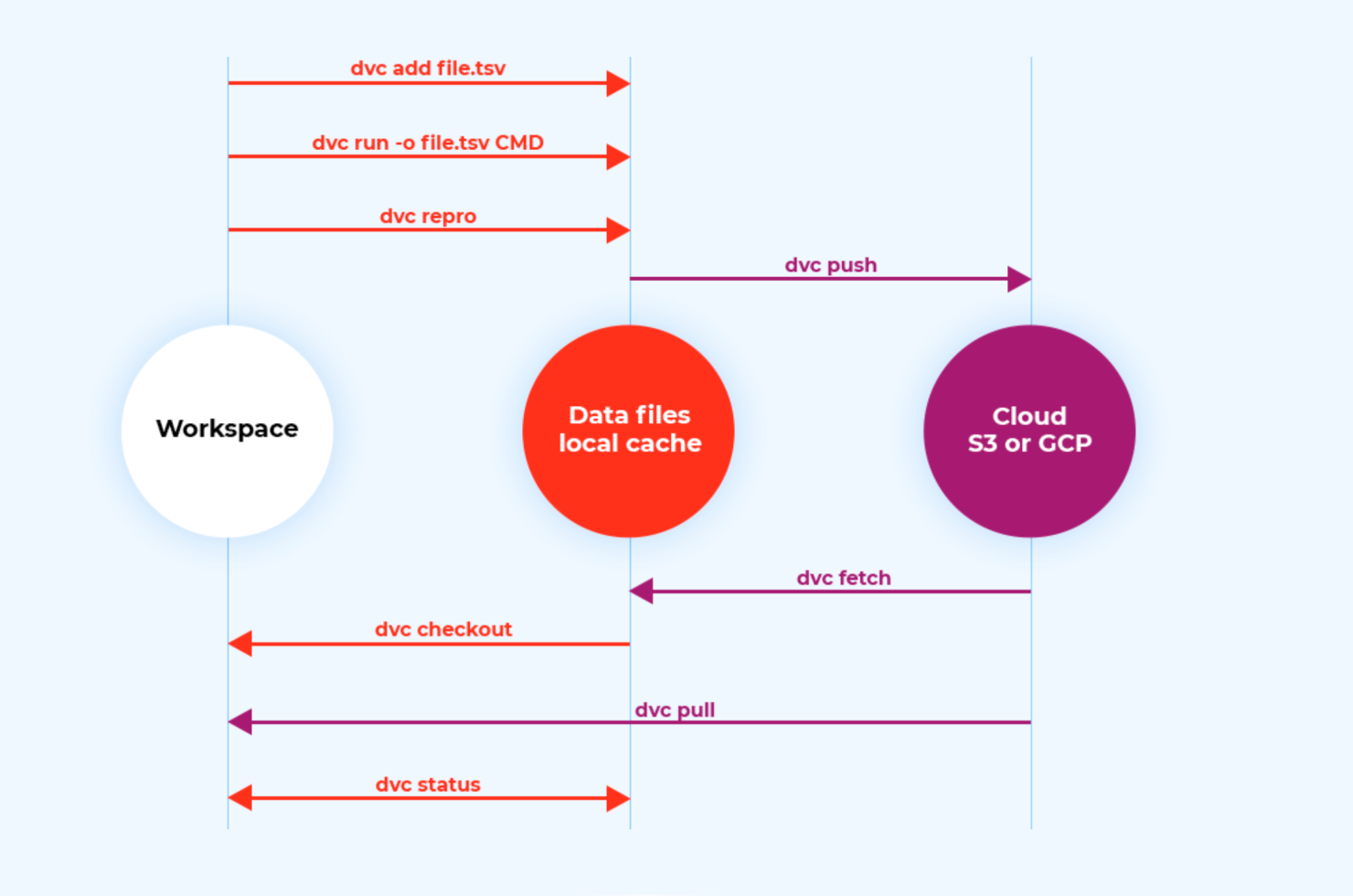
### 2. Sandbox environments

Data scientists often rave about Jupyter Notebooks, a sandbox-type environment that lets you run code in cells and insert Markdown in between (or at least I rave about them). Jupyter Notebooks are like writing a book with code in them: you can be detailed about what each cell does, and organize things in a visually pleasing way. Separating code into cells and sections is a viable way to version your different models.

When it comes to deployment and production though, versioning your models in a notebook doesn’t really cut it. Jupyter Notebooks are a tool for exploration and visualization, not for managing dependencies and tracking minute changes to hyperparameters.

### 3. Data Version Control (DVC)

Data Version Control (DVC) is a Git extension that adds functionality for managing your code and data together. It works directly with cloud storage (AWS S3 or Google GCP) to push your data changes. For example, according to their tutorial, “DVC streamlines large data files and binary models into a single Git environment and this approach will not require storing binary files in your Git repository.” It’s a streamlined version of combining Git with machine learning specific functionality for data management.



For a tutorial on how to implement DVC in your project and why it’s so helpful, [check out this walkthrough](https://becominghuman.ai/how-to-version-control-your-machine-learning-task-ii-d37da60ef570" \t "_blank).

### 4. Commercial solutions

The traditional business wisdom tells us that if there’s a problem, there’s a company. There are a few companies starting out attempting to solve the data versioning problem. [Comet.ml](https://www.comet.ml/) is an automatic versioning solution that tracks and organizes all of your team’s modeling efforts. You can easily compare experiments, see the differences in code between two models, and invite team members to collaborate on a project.

### 5. Platforms as a Service and Algorithmia

Even once you’ve found a way to manage data versioning during your training and experimentation process, much of the complexity resides in inference: deploying the right models in the right places at the right times. If you’re using a Platform as a Service to deploy your machine learning models, it might offer some functionality around data versioning.

If you’re deployed on the Algorithmia platform, we productionize your models as independent microservices with individual endpoints. That means you can continue to reference historical versions of your models in production without having to worry about them breaking or getting deprecated. It’s as simple as appending the model name in our API with a version number.