Venkatesh Pappakrishnan, Ph.D. Follow
Data Scientist | Physicist
Feb 10 · 11 min read

dreamstime/Scyther5

How to write a production-level code in Data Science?

Ability to write a production-level code is one of the sought-after skills for a data scientist role— either posted explicitly or not. For a software engineer turned data scientist this may not sound like a challenging task as they might have already perfected their skill at developing production level codes and deployed into production several times.

This article is for those who are new to writing production-level code and interested in learning it such as fresh graduates from universities or any professionals who made into data science (or planning to make the transition). For them writing production-level code might seem like a formidable task.

I will give you tips on how to write a production-level code and practice it, you don't necessarily have to be in a data science role to learn this skill.

1. Keep it modular

This is basically a software design technique recommended for any software engineer. The idea here is to break a large code into small independent sections(functions) based on its functionality. There are two parts to it.

(i) Break the code into smaller pieces each intended to perform a specific task (may include sub tasks)

(ii) Group these functions into modules (or python files) based on its usability. It also helps in staying organized and ease of code maintainability

The first step is to decompose a large code into many simple functions with specific inputs (and input formats) and outputs (and output formats). As mentioned earlier, each function should perform a single task such as *cleanup outliers in the data, replace erroneous values, score a model, calculate root-mean-squared error (RMSE)*, and so on. Try to break each of those functions further down to performing sub tasks and continue till none of the functions can be further broken down.

Low-level functions—the most basic functions that cannot be further decomposed. For example, computing *RMSE* or Z-score of the data. Some of these functions can be widely used for training and implementation of any algorithm or machine learning model.

Medium-level functions—a function that uses one or more of low-level functions and/or other medium-level functions to perform its task. For instance, *cleanup outliers* function use *compute Z-score* function to remove the outliers by only retained data within certain bounds or an *error* function that uses *compute RMSE* function to get RMSE values.

High-level functions—a function that uses one or more of medium-level functions and/or low-level functions to perform its task. For example, model training function that uses several functions such as function to get randomly sampled data, a model scoring function, a metric function, etc.

The final steps are to group all the low-level and medium-level functions that will be useful for more than one algorithm into a python file (can be imported as a module) and all other low-level and medium-level functions that will be useful only for the algorithm in consideration into another python file. All the high-level functions

should reside in a separate python file. This python file dictates each step in the algorithm development—from combining data from different sources to final machine learning model.

There is no hard-and-fast rule to follow the above steps but I highly suggest you to start with these steps and develop your own style there after.

2. Logging and Instrumentation

Logging and Instrumentation (LI) are analogous to black box in air crafts that record all the happenings in the cockpit. The main purpose of LI is to record useful information from the code during its execution to help the programmer mainly to debug if anything goes awry and also to improve the performance of the code (such as reduced execution times).

What is the difference between Logging and Instrumentation?

(i) Logging—Records only actionable information such as critical failures during run time and structured data such as intermediate results that will be later used by the code itself. Multiple log levels such as debug, info, warn, and errors are acceptable during development and testing phases. However avoid them at all cost during production.

Logging should be minimal containing only information that requires human attention and immediate handling.

(ii) Instrumentation—records all other information left out in logging that would help us validate code execution steps and work on performance improvements if necessary. Here it is always better to have more data so instrument as much information as possible.

To validate code execution steps—We should record information such as task name, intermediate results, steps went through, etc. This would help us to validate the results and also to confirm that the algorithm has followed the intended steps. Invalid results or strangely performing algorithm may not raise a critical error that would be caught in logging. Hence it is imperative to records these information.

To improve performance—We should record time taken for each task/subtask and memory utilized by each variable. This would help us

improve our code in making necessary changes optimizing the code to run faster and limit memory consumption (or identify memory leaks which is common in python).

Instrumentation should record all other information left out in logging that would help us to validate code execution steps and work on performance improvements. It is better to have more data than less.

3. Code Optimization

Code optimization implies both reduced time complexity (run time) as well as reduced space complexity (memory usage). The time/space complexity is commonly denoted as O(x) also known as ***Big-O representation*** where x is the dominant term in time- or space- taken polynomial. The time- and space- complexity are the metric for measuring ***algorithm efficiency***.

For example, lets say we have a nested *for* loop of size *n* each and takes about 2 seconds each run followed by a simple *for* loop that takes 4 seconds for each run. Then the equation for time consumption can be written as

Time taken $\sim 2\,n^2+4\,n = O(n^2+n) = O(n^2)$

For Big-O representation, we should drop the non-dominant terms (as it will be negligible as *n* tends to *inf*) as well as the coefficients. The coefficients or the scaling factors are ignored as we have less control over that in terms of optimization flexibility. Please note that the coefficients in the absolute time taken refers to the product of number of *for* loops and the time taken for each run whereas the coefficients in $O(n^2+n)$ represents the number of *for* loops (1 double *for* loop and 1 single *for* loop). Again we should drop the lower order terms from the equation. Hence the Big-O for the above process is $O(n^2)$.

Now, our goal is to replace least efficient part of the code with a better alternative with lower time complexity. For example, O(n) is better than $O(n^2)$. The most common killers in the code are *for* loops and the least common but worse than *for* loop are recursive functions (O(branch^depth)). Try to replace as many *for* loops as possible with python modules or functions which are usually heavily optimized with possible C-code performing the computation, instead of python, to achieve shorter run time.

I highly recommend you to read the section about “Big-O” in *Cracking the coding interview* by Gayle McDowell. In fact, try to read the entire book to improve your coding skills.

4. Unit Testing

Unit testing—automates code testing in terms of functionality

Your code have to clear multiple stages of testing and debugging before getting into production. Usually there are three levels—development, staging, and production. In some companies, there will be a level before production that mimics the exact environment of a production system. The code should be free from any obvious issues and should be able to handle potential exceptions when it reaches production.

To be able to identify different issues that may rise we need to test our code against different scenarios, different data sets, different edge and corner cases, etc. It is inefficient to carry out this process manually every time we want to test the code which would be every time we make a major change to the code. Hence opt for Unit testing which contains a set of test cases and it can be executed whenever we want to test the code.

We have to add different test cases with expected results to test our code. The unit testing module goes through each test case, one-by-one, and compares the output of the code with the expected value. If the expected results are not achieved, the test fails —it is an early indicator that you code would fail if deployed into production. We need to debug the code and then repeat the process until all test cases are cleared off.

To make our life easy, python has a module called *unittest* to implement unit testing.

5. Compatibility with ecosystem

Most likely, your code is not going to be a standalone function or module. It will to be integrated into company’s code ecosystem and your code has to run synchronously with other parts of the ecosystem without any flaws/failures.

For instance, lets say that you have developed an algorithm to give recommendations. The process flow usually consists of getting recent

data from the database, update/generate recommendations, store it in a database which will be read by front-end frameworks such as webpages (using APIs) to display the recommended items to the user. Simple! It is like a chain, the new chain-link should lock-in with the previous and the next chain-link otherwise the process fails. Similarly, each process has to run as expected.

Each process will have a well-defined input and output requirements, expected response time, and more. If and when requested by other modules for updated recommendations (from webpage), your code should return the expected values in a desired format in an acceptable time. If the results are unexpected values (suggesting to buy milk when we are shopping for electronics), undesired format (suggestions in the form of texts rather than pictures), and unacceptable time (no one waits for mins to get recommendations, at least these days)—implies that the code is not in sync with system.

The best way to avoid such scenario is to discuss with the relevant team about the requirements before we begin the development process. If the team is not available, go through the code documentation (most probably you will find a lot of information in there) and code itself, if necessary, to understand the requirements.

6. Version Control

Git—a version control system is one of the best things that has happened in recent times for source code management. It tracks the changes made to the computer code. Perhaps there are many existing version control/tracking systems but Git is widely used compared to any other.

The process in simple terms “modify and commit”. I have over simplified it. There are so many steps to the process such as creating a branch for development, committing changes locally, pulling files from remote, pushing files to remote branch, and much more which I am going to leave it to you to explore on your own.

Every time we make a change to the code, instead of saving the file with a different name, we commit the changes—meaning overwriting the old file with new changes with a key linked to it. We usually write comments every time we commit a change to the code. Let’s say, you

don't like changes made in the last commit and want to revert back to previous version, it can be done easily using the commit reference key. Git is so powerful and useful for code development and maintenance.

You might have already understood why this is important for production systems and why it is mandatory to learn Git. We must always have the flexibility to go back to an older version that is stable just in case the new version fails unexpectedly.

7. Readability

The code you write should be easily digestible for others as well, at least for your team mates. Moreover, it will be challenging even for you to understand your own code in few months after writing the code, if proper naming conventions are not followed.

(i) Appropriate variable and function names

The variable and function names should be self explanatory. When someone reads your code it should be easy for then to find what each variable contains and what each function does, at least to some extent.

It is perfectly okay to have a long name that clearly states its functionality/role rather than having short names such as x, y, z, etc., that are vague. Try not to exceed 30 char for variable names and 50–60 for function names.

Previously, the standard code width was 80 char based on IBM standard which is totally outdated. Now, as per GitHub standards, it is around 120. Setting 1/4th limit of page width for character names we get 30 which is long enough yet doesn't fill the page. The function names could be little longer but again shouldn't fill the entire page. So by setting a limit of 1/2th of page width we get 60.

For instance, the variable for average age of Asian men in a sample data can be written as *mean_age_men_Asia* rather than *age* or *x*. Similar argument applies for function names as well.

(ii) Doc string and comments

In addition to appropriate variable and function names, it is essential to have comments and notes wherever necessary to help the reader in

understanding the code.

Doc string—Function/class/module specific. The first few lines of text inside the function definition that describes the role of the function along with its inputs and outputs. The text should be placed between set of 3 double quotes.

```
def <function_name>:

"""<docstring>"""

return <output>
```

Comments—can be placed any where in the code to inform the reader about the action/role of a particular section/line. The need for comments will be considerable reduced if we give appropriate names to variables and functions—the code will be, for the most part, self explanatory.

Code review:

Although, it is not a direct step in writing production quality code, code review by your peers will be helpful in improving your coding skill.

No one writes a flawless computer code, unless someone has more than 10 years of experience. There will be always room for improvement. I have seen professionals with several years of experience writing an awful code and also interns who were pursuing their bachelors degree with outstanding coding skills—you can always find someone who is better than you. It all depends on how many how many hours someone invests in learning, practicing, and most importantly improving that particular skill.

I know that people better than you always exist but it is not always possible to find them in your team with only whom your can share your code. Perhaps you are the best in your team. In that case, it is okay to ask others in the team to test and give feedback to your code. Even though, they are not as good as you, something might have escaped your eyes that they might catch.

Code review is especially important when you are in early stages of your career. It would greatly improve your coding skills. Please follow

the steps below for successfully getting your code reviewed.

(i) After you complete writing your code with all the development, testing, and debugging. Make sure you don't leave out any silly mistakes. Then kindly request your peers for code review.

(ii) Forward them your code link. Don't ask them to review several scripts at one time. Ask them one after the other. The comments they give for the first script are perhaps applicable to other scripts as well. Make sure you apply those changes on other scripts, if applicable, before sending out the second script for review.

(iii) Give them a week or two to read and test your code for each iteration. Also provide all necessary information to test your code like sample inputs, limitations, and so on.

(iv) Meet with each one of them and get their suggestions. Remember, you don't have to included all their suggestions in your code, select the ones that you think will improve the code at your own discretion.

(v) Repeat until you and your team are satisfied. Try to fix or improve your code in the first few iterations (max 3–4) otherwise it might create a bad impression about your code ability.

Hope this article is helpful and you enjoyed reading it.

I would love to read your feedback.