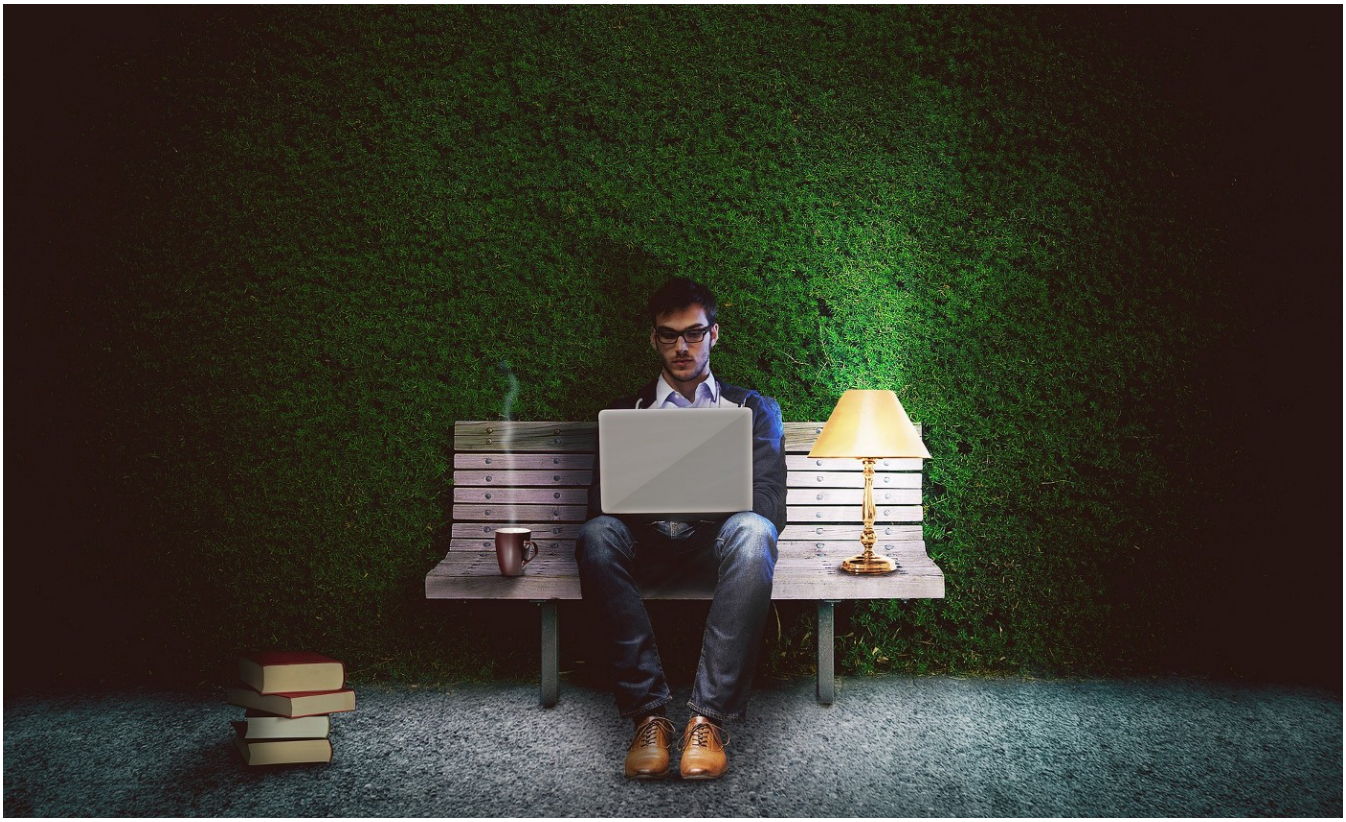

12 Steps to Production-Quality Data Science Code

Genevieve Hayes

Sep 1, 2019 · 14 min read ★

There's a Dilbert comic in which Dilbert tells his boss that he can't take over a co-worker's software project until he spends a week bad mouthing the co-worker's existing code. If you've ever taken over maintaining someone else's code, you'll immediately see the truth in this.

No one likes taking over maintaining or working on someone else's code. At best, you're faced with hours or days of trying to figure out what the previous programmer was thinking when they made certain decisions — even if the code was well commented.

Once, I inherited a 40+ page piece of SAS code, from a recently retired co-worker, that had so many dead ends, unnecessary variables, and commented out pieces of rubbish that, by the time I had cleaned it up, the final code was less than two pages long. Over five years later, I still haven't quite gotten over the experience.

I suspect it is impossible to write code that no one is ever going to complain about. However, it *is* possible to keep the number of things that people can complain about in your code to a minimum.

Do Data Scientists Even Need to Write Good Code?

If your code is going to be put into production, then it should be of "production-quality".

What is meant by going into "production" varies from job to job, but for the purposes of this article, let's define production code as the final version of a piece of work which will either be run on a production server using real data, or serve as evidence of a piece of analysis.

By this definition, the following are all examples of production code:

- Code contained in the final version of a Jupyter notebook detailing a piece of analysis;
- Code used to produce an automated monthly report or to perform automated data extraction/wrangling;
- Code used to run the final version of a machine learning pipeline or an app that sits over the top of that pipeline; and
- Code used to create a package in a programming language, such as Python or R.

These are all examples of tasks that data scientists are regularly required to perform, even though most data scientists are not involved in software engineering, and are all situations where the code produced by the data scientist will typically need to be shared and maintained into the future.

As a result, it is important that the code is of the highest quality. That is, "production

- Readable;
- Free from errors;
- Robust to exceptions;
- Efficient;
- Well documented; and
- Reproducible.

If you want to see an example of production-quality code, check out the GitHub repository for your favorite Python package (for example, scikit-learn or NumPy). But, how do you get your code to a production-quality standard?

Here is the exact 12 step process I used when developing my own Python package, mlrose.

Step 1: Determine what you're trying to achieve

Before you even write a line of code, work out exactly *why* you are writing that code. What is it you are hoping to achieve? That is, what is the *goal* of your code?

If you are performing a piece of analysis, what research questions are you trying to answer?

If you are building a machine learning model, what outputs are you hoping to produce and what are you going to do with them once you get them?

If you are writing a package for a programming language, what functionality do you want that package to have?

Once you've got an answer to this question, run it by any clients or stakeholders you may be working with, to make sure everyone is on the same page, and that you aren't just about to spend days/weeks/months of your life producing something that isn't actually wanted.

When everyone is in agreement (or at least the majority of people), that's when it's

Write a piece of code that achieves the goal you identified in Step 1. Don't worry about what it looks like or whether it is efficient or not. Your aim is simply to produce a working piece of code that does what you want it to do. You'll improve the quality of your code in the later steps of this process.

If there are any issues in achieving your Step 1 goals, this is where you want to identify them (and address them), before you have invested too much time in your code.

Step 3: Reduce repetition

Apply the DRY (Don't Repeat Yourself) software development principle to your code.

The aim of the DRY principle is to minimize repetition in your code, in order to make it more readable, easier to maintain and reduce the risk of errors.

The most common way in which the DRY principle is applied in practice is through the creation of function. If you write essentially the same code segment more than once, convert it into a function or functions (ideally, each function should do just one thing).

For example, applying the DRY principle, you could rewrite:

```
import math
area2 = math.pi*2**2
area5 = math.pi*5**2
area7 = math.pi*7**2
area10 = math.pi*10**2
```

as:

```
import math

def circle_area(radius):
    return math.pi*radius**2

area2 = circle_area(2)
area5 = circle_area(5)
area7 = circle_area(7)
area10 = circle_area(10)
```

Step 4: Create unit tests

For each function in your code, write one or more tests to ensure it's doing what it's meant to be doing, under a range of different circumstances. Such tests are called "unit tests" and they can be as simple or as complex as you want them to be.

When writing my own unit tests, I will typically come up with examples simple enough that I can determine the function output by hand/calculator/Microsoft Excel, then write unit tests around them.

For example, for the circle_area() function given in Step 3, this could involve checking that circle_area(2) does, in fact, output 12.56637 and that circle_area(0) outputs 0.

In Python, these tests can be automated, using the unittest package.

To run your unit tests through the unittest package, create a class, and write your unit tests as methods (i.e. functions) that sit within the class. For example, to test the circle_area() function, you could write:

```
import unittest

class TestFunctions(unittest.TestCase):
    def test_circle_area2(self):
        assert abs(circle_area(2) - 12.56637) < 0.0001

    def test_circle_area0(self):
        assert circle_area(0) == 0

unittest.main(argv=[''], verbosity=2, exit=False)
```

In this example, I've called my test class "TestFunctions", but you can call it anything you want, provided the class has unittest.TestCase as its parent class.

Within this class, I've created two unit tests, one to test circle_area() works for radius 2 and one for radius 0. The names of these functions, again, don't matter, except that they must start with test_ and have the parameter self.

The final line of the code runs the tests.

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Alternatively, if one of your tests fails, then the top line of the output will include an “F” for each failed test and further output will be provided, giving details of the failures.

```
F.
======================================================================
FAIL: test_circle_area0 (__main__.TestFunctions)
----------------------------------------------------------------------
Traceback (most recent call last):
    File "D:/Documents/Unit_Test_Examples.py", line 13, in
test_circle_area0
    assert circle_area(0) == 1
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

If you’re writing your code using Python scripts (i.e. .py files), ideally you should house your unit tests in a separate testing file, to keep them apart from your main code. However, if you’re using a Jupyter notebook, you can place the unit tests in the final cell of the notebook.

Once you’ve created your unit tests and got them working, it’s worthwhile re-running them whenever you make any (significant) changes to your code.

Step 5: Deal with exceptions

When you write a function, you typically write it with the expectation that the function parameters will take on values only of a certain type or within a certain range. For example, in the circle_area() function, previously described, it is implicitly assumed that the parameter inputs will be non-negative integers or floats.

Best case scenario, your code will fall over, but the error message may be unclear.

Worst case scenario, your code will run and produce a nonsense result, which you may not realize is nonsense at the time, causing flow-on problems down the line.

To deal with this issue, you can add exception tests to your functions to check that the parameter inputs are as expected and output a customized error message if they are not.

For example, we could expand the circle_area() function to include exception tests to verify that the value of radius is an integer or float (the first if statement) and non-negative (the second if statement).

```
def circle_area(radius):
    if not isinstance(radius, int) and not isinstance(radius,
float):
        raise Exception("""radius must be an integer or float.""")

    if radius < 0:
        raise Exception("""radius must be non-negative.""")

    return math.pi*radius**2
```

If a parameter value fails one of the exception tests, the exception message, given as the argument of Exception(), will be printed and the code will stop.

Step 6: Maximize efficiency

It's not good enough to have code that is just readable and error free (although, that's a good start), you also want it to be both time and space efficient. Afterall, does it really matter if your code contains no errors if it takes 27 years to run, or takes up more space than you have RAM in your computer?

So, it's worthwhile reviewing your code to see if there are any ways in which you can make it run faster or take up less space.

In the case of Python, a lot has been written about performance optimization. However, in my experience, simply going through your code and vectorizing any for loops using

For example, consider the problem of summing a vector containing one billion ones, which we have attempted to solve in two different ways, the first version using a for loop and the second version using the NumPy sum() function.

```
import numpy as np
import time

x = np.ones(1000000000)

# For loop version
t0 = time.time()

total1 = 0

for i in range(len(x)):
    total1 += x[i]

t1 = time.time()

print('For loop run time:', t1 - t0, 'seconds')

# Numpy version
t2 = time.time()

total2 = np.sum(x)

t3 = time.time()

print('Numpy run time:', t3 - t2, 'seconds')
```

Using a for loop, this problem takes 275 seconds (over 4 1/2 minutes) to solve. However, using the NumPy sum function, the run time reduces to 15.7 seconds. That is, the for loop takes 17.5 times longer than the NumPy summation.

```
For loop run time: 275.49414443969727 seconds
Numpy run time: 15.738749027252197 seconds
```

Step 7: Make names meaningful

Have you ever read code where every variable has a name like “x” or “temp” (and even

I've seen code like this, where I've been forced to guess the definitions of variables and then keep a handwritten list of these definitions in order to follow the code. That's not saying much about the code's readability.

You can solve this problem by reviewing your code and replacing any meaningless variable or function names with more descriptive alternatives.

For example, you could rewrite:

```
def area(l, w):
    return l*w
```

as:

```
def rectangle_area(length, width):
    return length*width
```

Step 8: Check against a style guide

A coding style guide is a document that sets out all the coding conventions and best practices for a particular programming language. In Python, the go to style guide is PEP 8 — Style Guide for Python Code which includes such advice as:

- "Limit all lines to a maximum of 79 characters;"
- "Function names should be lowercase, with words separated by underscores as necessary to improve readability;" and
- "Use inline comments sparingly… Inline comments are unnecessary and in fact distracting if they state the obvious."

PEP 8 is a 27 page document, so ensuring your code is compliant with every single item can be a chore. Fortunately, there are tools to assist with this.

If you are writing your code as a Python script, the flake8 package will check for PEP 8

After installing this package (using pip install flake8), just navigate to the folder containing the code you want to check and run the following command at the command prompt:

```
flake8 filename.py
```

The output will tell you exactly where your code is non-compliant.

For example, this output tells us that the Python script Production_Examples.py contains 8 instances of non-compliance. The first of these instances is in row 2, column 1, where the package 'numpy' is imported but unused:

```
Production_Examples.py:2:1: F401 'numpy as np' imported but unused
Production_Examples.py:66:1: E302 expected 2 blank lines, found 1
Production_Examples.py:76:14: E261 at least two spaces before inline
comment
Production_Examples.py:76:15: E262 inline comment should start with
'# '
Production_Examples.py:79:1: E302 expected 2 blank lines, found 1
Production_Examples.py:86:1: W293 blank line contains whitespace
Production_Examples.py:93:1: W293 blank line contains whitespace
Production_Examples.py:96:1: E305 expected 2 blank lines after class
or function definition, found 1
```

In Jupyter Notebooks, several extensions exist to ensure PEP 8 compliance, including jupyterlab-flake8 and jupyter-autopep8.

Step 9: Ensure reproducibility

Reproducibility means different things, depending on the task you are trying to perform.

If you are developing code to be run by others on a regular basis (for example, the code behind a regular report or a machine learning pipeline), then reproducibility can just mean establishing a master copy of your code that you can maintain and others can make use of. For example, by creating a GitHub repository for your project.

your analysis, so that any claims you make based on that analysis can be independently verified.

Many statistical and machine learning processes involve some degree of randomization. For example, randomly splitting a dataset into training and test subsets, or the randomized optimization aspects of many machine learning algorithms.

This means that every time these processes are rerun, slightly different outputs will be produced, even if the input data is exactly the same.

You can avoid this situation by setting the random seed, either at the start of your program (for example, in Python this can be done using the NumPy np.random.seed() function), or as a parameter in any algorithms involving randomization (for example, many Python scikit-learn functions include an optional random_state parameter for this purpose).

This will guarantee that, as long as the inputs to your program remain unchanged, the outputs will always be the same.

Step 10: Add comments and documentation

Write docstrings for all function, classes and methods, and add block or inline comments (that is, comments that are preceded by a #) to clarify any sections of your code where the purpose may not be immediately obvious.

Most programmers are already familiar with, and make use of, block and inline comments, as illustrated by the example below:

```
# This is a block comment
def test_function(x): y = x + 5 # This is an inline comment
    return y
```

However, docstrings are less commonly used.

A docstring is a summary of what a function (or class or method) does, enclosed in triple quotes. It will typically include the following (although, the last two items are

- A description of each of the function's arguments (including their expected types and any default values); and
- A description of each of the function's outputs (including type).

For example, a docstring for our circle_area() function (which I have modified slightly for illustration purposes) might look like:

```
def circle_area(radius=1):
    """Calculate the area of a circle with a given radius.
    Parameters
    ----------
    radius: float, default: 1
        The radius of the circle. Must be non-negative.

    Returns
    -------
    area: float
        The area of the circle.
    """
    area = math.pi*radius**2

    return area
```

The formatting convention used in this docstring example was deliberately chosen to integrate with the Python sphinx package.

Sphinx is commonly used to create the documentation for Python packages, and has been used to create the documentation for many popular packages, including NumPy, NLTK and scikit-learn), and can literally convert your docstrings into documentation in a variety of formats, including HTML, pdf and ePub.

Here is an example of documentation created using sphinx:

It also integrates with GitHub and ReadtheDocs (a documentation hosting platform), such that your documentation is automatically rebuilt whenever any updates to your code are pushed to GitHub, ensuring your documentation always stays up to date.

Step 11: Ask for a code review

If you work in an environment with other programmers, then you should always ask for a code review before deploying your work. Aside from the advantage of potentially being able to pick up any errors that may exist in your code, you can ask your reviewer for feedback on a variety of matters including:

- how clear your documentation is;
- the coverage of your unit tests;
- whether they can see scope for efficiency improvements that you may have missed; and
- whether your code achieves your initial goals.

But what if you're still early in your career and don't yet work in a programming team?

Some alternative approaches you can employ to get your code reviewed include:

- Making use of online forums where users can request code feedback from other users, including Code Review Stack Exchange and the review subreddits: /r/codereview/, /r/reviewmycode/ and /r/critiquemycode/; or

- If you are getting really desperate, and have no other alternative, putting your code aside for a period of time (say, a month) and then coming back to it when you have had time to forget the finer details and conducting your own review.

This step is one of the most important parts of the development process, since it is from code reviews that you will learn to become a better coder.

Step 12: Deploy

Once you have gone through all the previous steps, your code is ready to deploy. This doesn't mean your code isn't going to change again. It just means you have produced a piece of work that is of a standard you can be proud to put your name to.

In fact, once people start using your code, or its outputs, it's almost certain they will come back with improvements you never thought of, or detect errors you completely missed.

In anticipation of this, schedule some time now for one to six months into the future, where you will go back over your code once more and incorporate any feedback you have inevitably received post deployment.

A wise man by the name of John F. Woods once said:

> "Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

Having previously inherited code from other programmers who didn't know how to write production-quality code, I couldn't agree more. But even if your life doesn't depend on the quality of your code, producing good quality code should be a matter of professional pride.

that doesn't meet certain minimum coding standards, either.

That's what being a professional data scientist is all about.

Python Data Science Programming Towards Data Science