Good practices: coding practices, debugging, and reproducible research

Chris Paciorek

2024-09-03

Table of contents

PDF

[This Unit is under construction as of 2024-08-28.]

Sources:

- The Python PEP8 Style Guide
- Murrell, Introduction to Data Technologies, Ch. 2
- Journal of Statistical Software vol. 42: 19 Ways of Looking at Statistical Software
- Wilson et at., Best practices for scientific computing, ArXiv:1210:0530
- Gentzkow and Shapiro tutorial for social scientists
- Millman and Perez article about reproducible research
- Chapter 11 of Transparent and Reproducible Social Science Research

This unit covers good coding/software development practices, debugging (and practices for avoiding bugs), and doing reproducible research. As in later units of the course, the material is generally not specific to Python, but some details and the examples are in Python.

1. Good coding practices

Some of these tips apply more to software development and some more to analyses done for specific projects; hopefully it will be clear in most cases.

Editors

Use an editor that supports the language you are using (e.g., *Atom*, *Emacs*/*Aquamacs*, *Sublime*, *vim*, *VSCode*, *TextMate*, *WinEdt*, or the built-in editor in *RStudio* [you can use Python from within RStudio]). Some advantages of this can include:

1. helpful color coding of different types of syntax,
2. automatic indentation and spacing,
3. parenthesis matching,

4. line numbering (good for finding bugs), and
5. code can often be run (or compiled) and debugged from within the editor.

See the problem set submission how-to document for more information about editors that interact nicely with Quarto documents.

Code syntax and style

Coding syntax tips

The PEP 8 style guide is your go-to reference for Python style. I've highlighted some details here as well as included some general suggestions of my own.

- Header information: put metainfo on the code into the first few lines of the file as comments. Include who, when, what, how the code fits within a larger program (if appropriate), possibly the versions of Python and key packages that you used.
- Write docstrings for public modules, classes, functions, and methods. For non-public items, a comment after the `def` line is sufficient to describe the purpose of the item.
- Indentation: Python is strict about indentation of course, which helps to enforce clear indentation more than in other languages. This helps you and others to read and understand the code and can help in detecting errors in your code because it can expose lack of symmetry.
  - use 4 spaces per indentation level (avoid tabs if possible).
- Whitespace: use it in a variety of places. Some places where it is good to have it are
  - around operators (assignment and arithmetic);
  - between function arguments;
  - between list/tuple elements; and
  - between matrix/array indices.
- Use blank lines to separate blocks of code with comments to say what the block does.
- Use whitespaces or parentheses for clarity even if not needed for order of operations. For example, `a/y*x` will work but is not easy to read and you can easily induce a bug if you forget the order of ops. Instead, use `a/y * x`.
- Avoid code lines longer than 79 characters and comment/docstring lines longer than 72 characters.
- Comments: add lots of comments (but don't belabor the obvious, such as `x = x + 1  # increment x`).
  - Remember that in a few months, you may not follow your own code any better than a stranger.
  - Some key things to document: (1) summarizing a block of code, (2) explaining a very complicated piece of code - recall our complicated regular expressions, and (3) explaining arbitrary constant values.
  - Comments should generally be complete sentences.
- You can use parentheses to group operations such that they can be split up into lines and easily commented, e.g.,

```
newdf = (
        pd.read_csv('file.csv')
        .rename(columns = {'STATE': 'us_state'})  # adjust column names
        .dropna()                                 # remove some rows
        )
```

- For software development, break code into separate files (2000-3000 lines per file) with meaningful file names and related functions grouped within a file.
- Being consistent about the naming style for objects and functions is hard, but try to be consistent. PEP8 suggests:
  - Class names should be UpperCamelCase.
  - Function, method, and variable names should be snake_case, e.g., `number_of_its` or `n_its`.
  - Non-public methods and variables should have a leading underscore.
- Try to have the names be informative without being overly long.
- Don't overwrite names of objects/functions that already exist in Python. E.g., don't use `len`. That said, the namespace system helps with the unavoidable cases where there are name conflicts.
- Use active names for functions (e.g., `calc_loglik`, `calc_log_lik` rather than `loglik` or `loglik_calc`). The idea is that a function in a programming language is like a verb in regular language (a function *does* something), so use a verb to name it.
- Learn from others' code

This semester, someone will be reading your code - the GSI and and me when we look at your assignments. So to help us in understanding your code and develop good habits, put these ideas into practice in your assignments.

While not Python examples, the files goodCode.R and badCode.R in the `units` directory of the class repository provide examples of code written such that it does and does not conform to the general ideas listed above (leaving aside the different syntax of Python and R).

Coding style suggestions

This is particularly focused on software development, but some of the ideas are useful for data analysis as well.

- Break down tasks into core units
- Write reusable code for core functionality and keep a single copy of the code (using version control or at least with a reasonable backup strategy) so you only need to make changes to a piece of code in one place
- Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion (like the UNIX utilities)
- Write functions that take data as an argument and not lines of code that operate on specific data objects. Why? Functions allow us to reuse blocks of code easily for later use and for recreating an analysis (reproducible research). It's more transparent than sourcing a file of code because the inputs and outputs are specified formally, so you don't have to read through the code to figure out what it does.

- Functions should:
  - be modular (having a single task);
  - have meaningful name; and
  - have a doc string describing their purpose, inputs and outputs.
- Write tests for each function (i.e., unit tests)
- Don't hard code numbers - use variables (e.g., number of iterations, parameter values in simulations), even if you don't expect to change the value, as this makes the code more readable. For example, the speed of light is a constant in a scientific sense, but best to make it a variable in code: `speed_of_light = 3e8`
- Use lists or tuples to keep disparate parts of related data together
- Practice defensive programming (see also the discussion below on assertions)
  - check function inputs and warn users if the code will do something they might not expect or makes particular choices;
  - check inputs to *if*:
    - Note that in Python, an expression used as the condition of an `if` will be equivalent to `True` unless it is one of `False`, `0`, `None`, or an empty list/tuple/string.
  - provide reasonable default arguments;
  - document the range of valid inputs;
  - check that the output produced is valid; and
  - stop execution based on checks and give an informative error message.
- Try to avoid system-dependent code that only runs on a specific version of an OS or specific OS
- Learn from others' code
- Consider rewriting your code once you know all the settings and conditions; often analyses and projects meander as we do our work and the initial plan for the code no longer makes sense and the code is no longer designed specifically for the job being done.

Linting

Linting is the process of applying a tool to your code to enforce style.

Here we'll see how to use `ruff`. You might also consider `black`.

We'll practice with ruff with a small module we'll use next also for debugging.

First, we check for and fix syntax errors.

```
ruff check test.py
```

Then we ask ruff to reformat to conform to standard style.

```
cp test.py test-save.py   # Not required, just so we can see what `ruff` did.
ruff format test.py
```

Let's see what changed:

```
diff test-save.py test.py
```

```
#| echo: false
mv -f test-save.py test.py
```

Assertions, exceptions and testing

Assertions, exceptions and testing are critically important for writing robust code that is less likely to contain bugs.

Exceptions

You've probably already seen exceptions in action whenever you've done something in Python that causes an error to occur and an error message to be printed. Syntax errors are different from exceptions in that exceptions occur when the syntax is correct but the execution of the code results in some sort of error.

Exceptions can be a valuable tool for making your code handle different modes of failure (missing file, URL unreachable, permission denied, invalid inputs, etc.). You use them when you are writing code that is supposed to perform a task (e.g., a function that does something with an input file) to indicate that the task failed and the reason for that failure (e.g., the file was not there in the first place). In such case, you want your code to raise an exception and make the error message informative as possible, typically by handling the exception thrown by another function you call and augmenting the message. Another possibility is that your code detects a situation where you need to throw an exception (e.g., an invalid input to a function).

The other side of the coin happens when you want to write a piece of code that handles failure in a specific way, instead of simply giving up. For example, if you are writing a script that reads and downloads hundreds of URLs, you don't want your program to stop when any of them fails to respond (or do you?). You might want to continue with the rest of the URLs, and write out the failed URLs in a secondary output file.

Using `try-except` to continue execution

If you want some code to continue running even when it encounters an error, you can use `try-except`. This would often be done in code where you were running some workflow rather than in functions that you write for general purpose use (e.g., code in a package you are writing).

Suppose we have a loop and we want to run all the iterations even if the code for some iterations fail. We can embed the code that might fail in a `try` block and then in the `except` block, run code that will handle the situation when the error occurs.

```
for i in range(n):
    try:
        <some code that might fail>
        result[i] = <actual result>
    except:
        <what to do if the code fails>
        result[i] = None
```

Strategies for invoking and handling errors

Here we'll address situations that might arise when you are developing code for general purpose use (e.g., writing functions in a package) and need that code to invoke an error under certain circumstances or deal gracefully with an error occurring in some code that you are calling.