Good practices: coding practices, debugging, and reproducible research

Chris Paciorek

2024-09-03

Table of contents

Overview **2**

1. Good coding practices **2**
Editors . . . 2
Code syntax and style . . . 3
Coding syntax tips . . . 3
Coding style suggestions . . . 4
Linting . . . 5
Assertions, exceptions and testing . . . 6
Exceptions . . . 6
Assertions . . . 8
Testing . . . 9
Automated testing . . . 10
Version control . . . 11

2. Debugging and recommendations for avoiding bugs **12**
Basic debugging strategies . . . 12
Using pdb . . . 13
Using `breakpoint` . . . 13
Post-mortem debugging . . . 14
pdb commands . . . 15
Invoking pdb on a function or block of code . . . 16
Invoking pdb on a module . . . 16
Some common causes of bugs . . . 16
Tips for avoiding bugs and catching errors . . . 17
Practice defensive programming . . . 17
Catch run-time errors with `try/except` statements . . . 18
Maintain dimensionality . . . 20
Find and avoid global variables . . . 20
Miscellaneous tips . . . 21
Code review . . . 21

3. Tips for running analyses **21**

4. Reproducible research **21**
Some basic strategies for reproducible analyses 22
Formal tools 23

Overview

PDF

References:

- 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 what the relevant context(s) is/are.

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 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 helpful 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 R 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) 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 raising exceptions and 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.
- Ask others to review your code and help by reviewing other people's 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
```

```
All checks passed!
```

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
```

```
1 file reformatted
```

Let's see what changed:

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

```
1c1
< x=  7
---
> x = 7
3,5d2
```

```
< def myfun(x, y =0):
< # This is a toy function.
<       return x*y
6a4,6
> def myfun(x, y=0):
>     # This is a toy function.
>     return x * y
```

So we see that `ruff` fixed spacing and properly indented the comment line.

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 (i.e., a *run-time 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 a 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, but can also be useful 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.

A basic situation is when you want to detect a situation where you need to invoke an error (i.e., "throw an exception").

With `raise` you can invoke an exception. Suppose we need an input to be a positive number. We'll use Python's built-in `ValueError`, one of the various exception types that Python provides and that you could use. You can also create your own exceptions by subclassing one of Python's existing exception classes. (We haven't yet discussed classes and object-oriented programming, so don't worry if you're not sure about what that means.)

```
def myfun(val):
    if val <= 0:
        raise ValueError("`val` should be positive")

myfun(-3)
```

```
ValueError: `val` should be positive
```

Next let's consider cases where your function runs some code that might return an error.

We'd often want to catch the error using `try-except`. In some cases we would want to notify the user and then continue (perhaps falling back to a different way to do things or returning `None` from our function) while in others we might want to provide a more informative error message than if we had just let the error occur, but still have the exception be raised.

First let's see the case of continuing execution.

```
import os

def myfun(filename):
    try:
        with open(filename, "r") as file:
            text = file.read()
    except Exception as err:
        print(f"{err}\nCheck that the file `{filename}` can be found "\
              f"in the current path: `{os.getcwd()}`.")
        return None

    return(text.lower())
```

```
myfun('missing_file.txt')
```

```
[Errno 2] No such file or directory: 'missing_file.txt'
Check that the file `missing_file.txt` can be found in the current path: `/accounts/vis/paciorek/teac
```

Finally let's see how we can intercept an error but then "re-raise" the error rather than continuing execution.

```
import requests

def myfun(url):
    try:
        requests.get(url)
    except Exception as err:
        print(f"There was a problem accessing {url}. "\
              f"Perhaps it doesn't exist or the URL has a typo?")
        raise

myfun("http://missingurl.com")
```

```
There was a problem accessing http://missingurl.com. Perhaps it doesn't exist or the URL has a typo?

ConnectionError: HTTPConnectionPool(host='missingurl.com', port=80): Max retries exceeded with url: /
```

Assertions

Assertions are a quick way to raise a specific type of Exception (AssertionError). Assertions are useful for performing quick checks in your code that the state of the program is as you expect. They're primarily intended for use during the development process to provide "sanity checks" that specific conditions are true, and there are ways to disable them when you are running your code for production purposes (to improve performance). A common use is for verifying preconditions and postconditions (especially preconditions). One would generally only expect such conditions not to be true if there is a bug in the code. Here's an example of using the `assert` statement in Python, with a clear assertion message telling the developer what the problem is.

```
number = -42
assert number > 0, f"number greater than 0 expected, got: {number}"
## Produces this error:
## Traceback (most recent call last):
##   File "<stdin>", line 1, in <module>
## AssertionError: number greater than 0 expected, got: -42
```

Various operators/functions are commonly used in assertions, including

```
assert x in y
assert x not in y
assert x is y
```

```
assert x is not y
assert isinstance(x, <some_type>)
assert all(x)
assert any(x)
```

Assertions can also be quite useful in the context of data analysis pipelines for checking that the state of the analysis is as expected. E.g., at various points, you might check that the dimension of your data is as expected, check for missing or extreme/unexpected values, etc.

Next we'll see that assertions are a core part of setting up tests.

Testing

Testing is informally what you do after you write some code and want to check that it actually works. But when you are developing important code (e.g. functions that are going to be used by others) you typically want to encode your tests in code. There are many reasons to do that, including making sure that if anyone changes the code later on and breaks something, the test suite should immediately indicate that.

Some people even advocate for writing a preliminary test suite before writing the code itself(!) as it can be a good way to organize work and track progress, as well as act as a secondary form of documentation for clarity. This can include tests that your code provides correct and useful errors when something goes wrong (so that means that a test might be to see if problematic input correctly produces an error). *Unit tests* are intended to test the behavior of small pieces (units) of code, generally individual functions. Unit tests naturally work well with the ideas above of writing small, modular functions. I recommend the `pytest` package, which is designed to make it easier to write sets of good tests.

Here is an example of the contents of a test file, called `test_dummy.py`.

```
import pytest
import numpy as np

import dummy

def test_numeric():
    assert dummy.add_one(3) == 4

# This test will fail.
def test_numpy_array():
    assert np.all(np.equal(dummy.add_one(np.array([3,4])), np.array([4,5])))

def test_bad_input():
    with pytest.raises(TypeError):
        dummy.add_one('hello')

def test_warning():
    with pytest.warns(UserWarning, match='complex'):
        dummy.add_one(1+3j)
```

We can then run the tests via `pytest` like this:

```
pytest mytestfile.py
```

```
======================================== test session starts ========================================
platform linux -- Python 3.12.2, pytest-8.1.1, pluggy-1.4.0
rootdir: /accounts/vis/paciorek/teaching/243fall24/fall-2024/units
plugins: anyio-4.3.0
collected 4 items

test_dummy.py .F..                                                                          [100%]

============================================= FAILURES ==============================================
_____________________________________ test_numpy_array _____________________________________

    def test_numpy_array():
>       assert np.all(np.equal(dummy.add_one(np.array([3,4])), np.array([4,5])))

test_dummy.py:11:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

x = array([3, 4])

    def add_one(x):
        if not isinstance(x, (float, int, complex)):
>           raise TypeError(f"`{x}` should be numeric")
E           TypeError: `[3 4]` should be numeric

dummy.py:5: TypeError
==================================== short test summary info ====================================
FAILED test_dummy.py::test_numpy_array - TypeError: `[3 4]` should be numeric
================================== 1 failed, 3 passed in 0.84s ==================================
```

In lab, we'll go over assertions, exceptions, and testing in detail.

Automated testing

Continuous integration (CI) is the term for carrying out actions automatically as your code changes. The most common kind of CI is automated testing - running your tests on your code when you make changes to the code. This enforces the discipline of running tests regularly and avoids the common problem that testing passes locally on your own machine, but fails for various reasons when done elsewhere.

A standard way to do this is via GitHub Actions (GHA).

To set up a GitHub Actions workflow, one

- specifies when the workflow will run (e.g., when a push or pull request is made, or only manually)
- provides instructions for how to set up the environment for the workflow

- provides the operations that the workflow should run.

The workflow is specified using a YAML file placed in the `.github/workflows` directory of the repository.

With GHA, you specify the operating system and then the steps to run in the YAML file. Some steps will customize the environment as the initial steps and then additional step(s) will run shell or other code to run your workflow. You use pre-specified operations (called *actions*) to do common things (such as checking out a GitHub repository and installing commonly used software).

When triggered, GitHub will run the steps in a virtual machine, which is called the *runner*.

Here's an example YAML file for testing an example package called `mytoy`:

```
on:
  push:
    branches:
    - main

jobs:
  CI:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    # Install package and dependencies
    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version: 3.12

    - name: Install mytoy and pytest
      run: |
        pip install pytest
        pip install --user .

    - name: Run tests
      run: |
        cd mytoy
        pytest
```

You'll use GitHub Actions to automate testing during your projects. Unfortunately, GitHub Actions is not available via `github.berkeley.edu` so we can't use it with your class repositories for your problem set work.

Version control

- Use it! Even for projects that only you are working on. It's the closest thing you'll get to having a time machine!

- Use an issues tracker (e.g., the GitHub issues tracker is quite nice), or at least a simple to-do file, noting changes you'd like to make in the future.
- In addition to good commit messages, it's a good idea to keep good running notes documenting your projects.

We'll be discussing Git a lot separately.

2. Debugging and recommendations for avoiding bugs

Python's `pdb` package provides a standard debugger, with the ipython `ipdb` being a useful wrapper around `pdb`. JupyterLab also has a debugger. Perhaps the most commonly-used graphical debugger is provided via VS Code, which I encourage you to check out, particularly if you already use VS Code.

Basic debugging strategies

Here we'll discuss some basic strategies for finding and fixing bugs. Other useful locations for tips on debugging include:

- Efficient Debugging by Goldspink
- Debugging for Beginners by Brody

Read and think about the error message (the *traceback*), starting from the bottom of the traceback. Sometimes it's inscrutable, but often it just needs a bit of deciphering. Looking up a given error message by simply doing a web search with the exact message in double quotes can be a good strategy, or you could look specifically on Stack Overflow.

Below we'll see how one can view the stack trace. Usually when an error occurs, it occurs in a function call that is nested in a series of function calls. This series of calls is the *call stack* and the *traceback* or *stack trace* shows that series of calls that led to the error. To debug, you'll often need to focus on the function being executed at the time the error occurred (which will be at the top of the call stack but the bottom of the traceback) and the arguments passed into that function. However, if the error occurs in a function you didn't write, the problem will often be with the arguments that your code provided at the last point in the call stack at which code that you wrote was run. Check the arguments that your code passed into that first function that is not a function of yours.

When running code that produces multiple errors, fix errors from the top down - fix the first error that is reported, because later errors are often caused by the initial error. It's common to have a string of many errors, which looks daunting, caused by a single initial error.

Is the bug reproducible - does it always happen in the same way at at the same point? It can help to restart Python and see if the bug persists - this can sometimes help in figuring out if there is a scoping issue and we are using a global variable that we did not mean to.

If you can't figure out where the error occurs based on the error messages, a basic strategy is to build up code in pieces (or tear it back in pieces to a simpler version). This allows you to isolate where the error is occurring. You might use a binary search strategy. Figure out which half of the code the error occurs in. Then split the 'bad' half in half and figure out which half the error occurs in. Repeat until you've isolated the problem.

If you've written your code modularly with lots of functions, you can test individual functions. Often the error will be in what gets passed into and out of each function.

At the beginning of time (the 1970s?), the standard debugging strategy was to insert print statements in one's code to see the value of a variable and thereby decipher what could be going wrong. We have better tools nowadays. But sometimes we still need to fall back to inserting print statements.

Python is a scripting language, so you can usually run your code line by line to figure out what is happening. This can be a decent approach, particularly for simple code. However, when you are trying to find errors that occur within a series of many nested function calls or when the errors involve variable scoping (how Python looks for variables that are not local to a function), or in other complicated situations, using formal debugging tools can be much more effective. Finally, if the error occurs inside of functions provided by Python, rather than ones you write, it can be hard to run the code in those functions line by line.

Using pdb

We can activate the debugger in various ways:

- by inserting `breakpoint()` (or equivalently `import pdb; pdb.set_trace()`) inside a function or module at a location of interest (and then running the function or module)
- by using `pdb.pm()` after an error (i.e., an *exception*) has occurred to invoke the browser at the point the error occurred
  - alternatively in IPython/Jupyter Notebook, run `%debug` (an IPython 'magic' command) and then run the code that results in the error
- by running a function under debugger control with `pdb.run()`
- by starting python with `python -m pdb file.py` and adding breakpoints

If you're using IPython or a Jupyter Notebook, `ipdb` is a wrapper for `pdb` that has all the same commands, but provides some nice features for interactivity (such as tab completion and syntax highlighting).

Using `breakpoint`

Let's define a function that will run a stratified analysis, in this case fitting a regression to each of the strata (groups/clusters) in some data. Our function is in `run_with_break.py`, and it contains `breakpoint` at the point where we want to invoke the debugger.

Now I can call the function and will be put into debugging mode just before the next line is called:

```
import run_with_break as run
run.fit(run.data, run.n_cats)
```

When I run this, I see this:

```
>>> run.fit(data, n_cats)
> /accounts/vis/paciorek/teaching/243/fall-2024/units/run_with_break.py(10)fit()
-> sub = data[data['cats'] == i]
(Pdb)
```

This indicates I am debugging at line 10 of `run_with_break.py`, which is the line that creates `sub`, but I haven't yet created `sub`.

I can type `n` to run that line and go to the next one:

```
(Pdb) n
> /accounts/vis/paciorek/teaching/243/fall-2024/units/run_with_break.py(11)fit()
-> model = statsmodels.api.OLS(sub['y'], statsmodels.api.add_constant(sub['x']))
```

at which point the debugger is about to execute line 11, which fits the regression.

I can type `c` to continue until the next breakpoint:

```
(Pdb) c
> /accounts/vis/paciorek/teaching/243/fall-2024/units/run_with_break.py(10)fit()
-> sub = data[data['cats'] == i]
```

Now if I print `i`, I see that it has incremented to `1`.

```
(Pdb) p i
1
```

We could keep hitting `n` or `c` until hitting the stratum where an error occurs, but that would be tedious.

Let's hit `q` to quit out of the debugger.

```
(Pdb) q
>>>
```

Note that all of these commands have graphical-based equivalents in VS Code.

Next let's see how we can enter debugging mode only at point an error occurs.

Post-mortem debugging

We'll use a version of the module without the `breakpoint()` command.

```
import pdb
import run_no_break as run

run.fit(run.data, run.n_cats)
pdb.pm()
```

That puts us into debugging mode at the point the error occurred:

```
> /usr/local/linux/miniforge-3.12/lib/python3.12/site-packages/numpy/core/fromnumeric.py(86)_wrapredu
-> return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
(Pdb)
```

which turns out to be in some internal Python function that calls a `reduce` function, which is where the error occurs (presumably the debugger doesn't enter this function because it calls compiled code):

```
(Pdb) l
 81                 if dtype is not None:
 82                     return reduction(axis=axis, dtype=dtype, out=out, **passkwargs)
```

```
 83                 else:
 84                     return reduction(axis=axis, out=out, **passkwargs)
 85
 86  ->     return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
 87
 88
 89     def _take_dispatcher(a, indices, axis=None, out=None, mode=None):
 90         return (a, out)
 91
```

We can enter `u` multiple times (it's only shown once below) to go up in the stack of function calls until we recognize code that we wrote:

```
(Pdb) u
> /accounts/vis/paciorek/teaching/243/fall-2024/units/run_no_break.py(10)fit()
-> model = statsmodels.api.OLS(sub['y'], statsmodels.api.add_constant(sub['x']))
```

Now let's use `p` to print variable values to understand the problem:

```
(Pdb)  p i
29
(Pdb) p sub
Empty DataFrame
Columns: [y, x, cats]
Index: []
```

Ah, so in the 29th stratum there are no data!

pdb commands

Here's a list of useful pdb commands (some of which we saw above) that you can use once you've entered debugging mode.

- `h` or `help`: shows all the commands
- `l` or `list`: show the code around where the debugger is currently operating
- `c` or `continue`: continue running the code until the next breakpoint
- `p` or `print`: print a variable or the result of evaluating an expression
- `n` or `next`: run the current line and go to the next line in the current function
- `s` or `step`: jump (step) into the function called in the current line (if it's a Python function)
- `r` or `run`: exit out of the current function (e.g., if you accidentally stepped into a function) (but note this stops at breakpoints)
- `unt` or `until`: run until the next line (or `unt <number>` to run until reaching line number); this is useful for letting a loop run until completion
- `b` or `break`: set a breakpoint
- `tbreak`: one-time breakpoint
- `where`: shows call stack
- `u` (or `up`) and `d` (or `down`): move up and down the call stack
- `q` quit out of the debugger
- `<return>`: runs the previous pdb command again

Invoking pdb on a function or block of code

We can use `pdb.run()` to run a function under the debugger. We need to make sure to use `s` as the first pdb command in order to actually step into the function. From there, we can debug as normal as if we had set a breakpoint at the start of the function.

```
import run_with_break as run
import pdb
pdb.run("run.fit(run.data, run.n_cats)")
(Pdb) s
```

Invoking pdb on a module

We can also invoke pdb when we start Python, executing a file (module). Here we've added `fit(data, n_cats)` at the end of `run_no_break2.py` so that we can have that run under the debugger.

```
#| eval: false
python -m pdb run_no_break2.py
```

```
> /accounts/vis/paciorek/teaching/243/fall-2024/units/run_no_break2.py(1)<module>()
-> import numpy as np
(Pdb)
```

Let's set a breakpoint at the same place we did with `breakpoint()` but using a line number (this avoids having to actually modify our code):

```
(Pdb) b 9
Breakpoint 1 at /accounts/vis/paciorek/teaching/243/fall-2024/units/run_no_break.py:9
```

```
(Pdb) c
> /accounts/vis/paciorek/teaching/243/fall-2024/units/run_no_break2.py(9)fit()
-> model = statsmodels.api.OLS(sub['y'], statsmodels.api.add_constant(sub['x']))
```

So we've broken at the same point where we manually added `breakpoint()` in `run_with_break.py`.

Or we could have set a breakpoint at the start of the function:

```
(Pdb) disable 1
Disabled breakpoint 1 at /accounts/vis/paciorek/teaching/243/fall-2024/units/run_no_break2.py:9
(Pdb) b fit
Breakpoint 1 at /accounts/vis/paciorek/teaching/243/fall-2024/units/run_no_break.py:6
```

Some common causes of bugs

Some of these are Python-specific, while others are common to a variety of languages.

- Parenthesis mis-matches
- `==` vs. `=`
- Comparing real numbers exactly using `==` is dangerous because numbers on a computer are only represented to limited numerical precision. For example,

```
1/3 == 4*(4/12-3/12)
```

```
False
```

We'll discuss in detail in Unit 8.

- You expect a single value but execution of the code gives an array
- Silent type conversion when you don't want it, or lack of coercion where you're expecting it
- Using the wrong function or variable name
- Giving unnamed arguments to a function in the wrong order
- Forgetting to define a variable in the environment of a function and having Python, via lexical scoping, get that variable as a global variable from one of the enclosing scope. At best the types are not compatible and you get an error; at worst, you use a garbage value and the bug is hard to trace. In some cases your code may work fine when you develop the code (if the variable exists in the enclosing environment), but then may not work when you restart Python if the variable no longer exists or is different.
- Python (usually helpfully) drops matrix and array dimensions that are extraneous. This can sometimes confuse later code that expects an object of a certain dimension. More on this below.

Tips for avoiding bugs and catching errors

Practice defensive programming

When writing functions, and software more generally, you'll want to warn the user or stop execution when there is an error and exit gracefully, giving the user some idea of what happened. Here are some things to consider:

- check function inputs and warn users if the code will do something they might not expect or makes particular choices;
- check inputs to `if` and the ranges in `for` loops;
- provide reasonable default arguments;
- document the range of valid inputs;
- check that the output produced is valid; and
- stop execution based on assertions, `try` or `raise` with an informative error message.

Here's an example of building a robust square root function:

```
import warnings

def mysqrt(x):
    if isinstance(x, str):
        raise TypeError(f"What is the square root of '{x}'?")
    if isinstance(x, (float, int)):
        if x < 0:
            warnings.warn("Input value is negative.", UserWarning)
            return float('nan')   # Avoid complex number result.
        else:
```

```
            return x**0.5
    else:
        raise ValueError(f"Cannot take the square root of {x}")


mysqrt(3.1)
mysqrt(-3)
```

```
/tmp/ipykernel_2412907/18181922.py:8: UserWarning:

Input value is negative.
```

```
nan
```

```
mysqrt('hat')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in mysqrt
TypeError: What is the square root of 'hat'?
```

Catch run-time errors with `try/except` statements

Also, sometimes a function you call will fail, but you want to continue execution. For example, consider the stratified analysis show previously in which you take subsets of your data based on some categorical variable and fit a statistical model for each value of the categorical variable. If some of the subsets have no or very few observations, the statistical model fitting might fail. To do this, you might be using a for loop or `apply`. You want your code to continue and fit the model for the rest of the cases even if one (or more) of the cases cannot be fit. You can wrap the function call that may fail within the `try` statement and then your code won't stop, even when an error occurs. Here's a toy example.

```
import numpy as np
import pandas as pd
import random
import statsmodels.api

np.random.seed(2)
n_cats = 30
n = 80
y = np.random.normal(size=n)
x = np.random.normal(size=n)
cats = [np.random.randint(0, n_cats-1) for _ in range(n)]
data = pd.DataFrame({'y': y, 'x': x, 'cats': cats})

params = np.full((n_cats, 2), np.nan)
for i in range(n_cats):
    sub = data[data['cats'] == i]
    try:
```

```
        model = statsmodels.api.OLS(sub['y'], statsmodels.api.add_constant(sub['x']))
        fit = model.fit()
        params[i, :] = fit.params.values
    except Exception as error:
        print(f"Regression cannot be fit for stratum {i}.")

print(params)
```

```
Regression cannot be fit for stratum 7.
Regression cannot be fit for stratum 20.
Regression cannot be fit for stratum 24.
Regression cannot be fit for stratum 29.
[[ 5.52897442e-01  2.61511154e-01]
 [ 5.72564369e-01  4.37210543e-02]
 [-9.91086764e-01  2.84116572e-01]
 [-6.50606465e-01  4.26310060e-01]
 [-2.59058826e+00 -2.59058826e+00]
 [ 8.59455139e-01 -4.64514288e+00]
 [ 3.82737032e-06  3.82737032e-06]
 [            nan             nan]
 [-5.55478634e-01 -1.17864561e-01]
 [-9.11601460e-02 -5.91519525e-01]
 [-7.30270153e-01 -1.99976841e-01]
 [-1.14495705e-01 -3.06421213e-02]
 [ 4.01648095e-01  9.30890661e-01]
 [ 7.88388728e-01 -1.45835443e+00]
 [ 4.08462508e+01  6.89262864e+01]
 [ 2.95467536e-01  8.80528901e-01]
 [ 1.04592517e+00  4.55379445e+00]
 [ 6.99549010e-01 -5.17503241e-01]
 [-1.75642254e+00 -8.07798224e-01]
 [-4.49033150e-02  3.53455362e-01]
 [            nan             nan]
 [ 2.63097970e-01  2.63097970e-01]
 [ 1.13328314e+00 -1.39985074e-01]
 [ 1.17996663e+00  3.68770563e-01]
 [            nan             nan]
 [-3.85101497e-03 -3.85101497e-03]
 [-8.04536124e-01 -5.19470059e-01]
 [-5.19200779e-01 -1.39952387e-01]
 [-9.16593858e-01 -2.67613324e-01]
 [            nan             nan]]
```

The stratum with id 7 had no observations, so that call to do the regression failed, but the loop continued because we 'caught' the error with `try`. In this example, we could have checked the sample size for the subset before doing the regression, but in other contexts, we may not have an easy way to

check in advance whether the function call will fail.

Maintain dimensionality

Python (usually helpfully) drops array dimensions that are extraneous. This can sometimes confuse later code that expects an object of a certain dimension. Here's a work-around:

```
import numpy as np
mat = np.array([[1, 2], [3, 4]])
np.sum(mat, axis=0)         # This sums columns, as desired

row_subset = 1
mat2 = mat[row_subset, :]
np.sum(mat2, axis=0)        # This sums the elements, not the columns.

if len(mat2.shape) != 2:    # Fix dimensionality.
    mat2 = mat2.reshape(1, -1)


np.sum(mat2, axis=0)
```

```
array([3, 4])
```

In this simple case it's obvious that a dimension will be dropped, but in more complicated settings, this can easily occur for some inputs without the coder realizing that it may happen. Not dropping dimensions is much easier than putting checks in to see if dimensions have been dropped and having the code behave differently depending on the dimensionality.

Find and avoid global variables

In general, using global variables (variables that are not created or passed into a function) results in code that is not robust. Results will change if you or a user modifies that global variable, usually without realizing/remembering that a function depends on it.

One ad hoc strategy is to remove objects you don't need from Python's global scope, to avoid accidentally using values from an old object via Python's scoping rules. You can also run your function in a fresh session to see if it's unable to find variables.

```
del x   # Mimic having a fresh sesson (knowing in this case `x` is global).

def f(z):
    y = 3
    print(x + y + z)

try:
    f(2)
except Exception as error:
    print(error)
```

```
name 'x' is not defined
```

Miscellaneous tips

- Use core Python functionality and algorithms already coded. Figure out if a functionality already exists in (or can be adapted from) an Python package (or potentially in a C/Fortran library/package): code that is part of standard mathematical/numerical packages will probably be more efficient and bug-free than anything you would write.
- Code in a modular fashion, making good use of functions, so that you don't need to debug the same code multiple times. Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion (like the UNIX utilities).
- Write code for clarity and accuracy first; then worry about efficiency. Write an initial version of the code in the simplest way, without trying to be efficient (e.g., you might use for loops even if you're coding in Python); then make a second version that employs efficiency tricks and check that both produce the same output.
- Plan out your code in advance, including all special cases/possibilities.
- Write tests for your code early in the process.
- Build up code in pieces, testing along the way. Make big changes in small steps, sequentially checking to see if the code has broken on test case(s).
- Be careful that the conditions of `if` statements and `while` loops and the sequences of `for` loops are robust.
- 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 and reduces bugs when you use the same number multiple times; e.g. `speed_of_light = 3e8` or `n_its = 1000`.

In a future Lab, we'll go over debugging in detail.

Code review

It's a good idea to have others review your code, ideally during the development process.

This article provides guidance on good practices for code review.

3. Tips for running analyses

Save your output at intermediate steps (including the random seed state) so you can restart if an error occurs or a computer fails. Using `pickle.dump()` to write to pickle files works well for this.

Run your code on a small subset of the problem before setting off a job that runs for hours or days. Make sure that the code works on the small subset and saves what you need properly at the end.

4. Reproducible research

The idea of "reproducible research" has gained a lot of attention in the last decade because of the increasing complexity of research projects, lack of details in the published literature, failures in being able to replicate or reproduce others' work, fraudulent research, and for other reasons.

We've seen a number of tools that can help with doing reproducible research, including version control systems such as git, the use of scripting such as bash and Python scripts, and literate programming tools such as Quarto and Jupyter notebooks.

Provenance is becoming increasingly important in science. It basically means being able to trace the steps of an analysis back to its origins. *Reproducibility* and *replicability* are related concepts:

Reproducibility - the idea is that a second person/group could get the exact same results as an existing analysis if they use the same input data, methods, and code. This can be surprisingly hard as time passes even if you're the one attempting to reproduce things.

Replicability - the idea is that a second/person could obtain results consistent with an existing analysis when using new data to answer the same scientific question.

Open question: What is required for something to be reproducible? What about replicable? What are the challenges in doing so?

Some basic strategies for reproducible analyses

- Have a directory for each project with subdirectories with meaningful and standardized names: e.g., `code`, `data`, `paper`. The Journal of the American Statistical Association (JASA) has a template GitHub repository with some suggestions.
- Have a file of code for pre-processing, one or more for analysis, and one for figure/table preparation.
  - The pre-processing may involve time-consuming steps. Save the output of the pre-processing as data file(s) that can be read in to the analysis script.
  - You may want to name your files something like this, so there is an obvious ordering: "1-prep.py", "2-analysis.py", "3-figs.py".
  - Have the code file for the figures produce the **exact** manuscript/report figures, operating on a file (e.g., a pickle file) that contains all the objects necessary to run the figure-producing code; the code producing the pickle file should be in your analysis code file (or somewhere else sensible).
  - Alternatively, use Quarto or Jupyter notebooks for your document preparation.
- Keep a document describing your running analysis with dates in a text file (i.e., a lab book).
- Note where data were obtained (and when, which can be helpful when publishing) and pre-processing steps in the lab book. Have data version numbers with a file describing the changes and dates (or in lab book). If possible, have all changes to data represented as code that processes the data relative to a fixed baseline dataset.
- Note what code files do what in the lab book.
- Keep track of the details of the system and software you are running your code under, e.g., operating system version, software (e.g., Python, R) versions, Python or R package versions, etc.
  - `pip list` and `conda list` will show you version numbers for installed packages.
  - `pip freeze > requirements.txt` and `conda env export > env.yml` will create a recipe file for your environment.
  - `pip install -r requirements.txt` and `conda env create -f env.yml` will build that environment on your machine.

Formal tools

1. In some cases you may be able to carry out your complete workflow in a Quarto document or in a Jupyter notebook.
2. You might consider workflow/pipeline management software such as Drake or other tools discussed in the CRAN Reproducible Research Task View. Alternatively, one can use the *make* tool, which is generally used for compiling code, as a tool for reproducible research: if interested, see the tutorial on Using make for workflows or this Journal of Statistical Software article for more details.
3. You might organize your workflow as a Python or R package as described (for the R case) in this article.
4. Package management:
    - Python: You can manage the versions of Python packages (and dependent packages) used in your project using Conda environments (or virtualenvs).
    - R: You can manage the versions of R packages (and dependent packages) used in your project using package management packages such as `renv` and `packrat`. Unfortunately, the useful `checkpoint` package relies on snapshots of CRAN that are not available after January 2023.
5. If your project uses multiple pieces of software (e.g., not just Python or R), you can set up a reproducible environment using *containers*, of which Docker containers are the best known. These provide something that is like a lightweight virtual machine in which you can install exactly the software (and versions) you want and then share with others. Docker container images are a key building block of various tools such as GitHub Actions and the Binder project. Alternatively (and often much easier) Conda is a general package manager that can install lots of non-Python packages and can also be used in many circumstances.