CODE QUALITY

Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.

-- <http://wiki.c2.com/?CodeForTheMaintainer>

As you work professionally, you'll have to maintain (fix bugs, add features) to existing code, and other people will do the same to your code. So writing maintainable, good quality, code is a necessary skill to have. During this class, work on moving from writing "studenty" code to professional quality code.

**Some features of quality code**

**Documentation**: for functions, methods, classes, modules. What do these things do? What does someone else need to know to use or understand this function, class, or module? Write documentation using your organization's preferred style; and follow formatting rules so that the code can be read by a documentation tool (e.g. Sphinx for Python, or JavaDoc for Java) which can automatically create a set of web pages documenting the project.

**Comments**: distinct from documentation - comments explain the logic of your code. You don't need to comment everything. Focus on the most complex and unintuitive parts of code.

**Descriptive names**: practice using descriptive names for functions, variables, modules. Follow a consistent naming style. The pep8 <https://www.python.org/dev/peps/pep-0008/>  style guide for Python suggests snake\_case for functions and UpperCamelCase for classes, and has various suggestions on how to name things.  Other languages follow different styles.

Some more tips on meaningful, but concise, variable names at: <http://deviq.com/naming-things/>  and <https://a-nickels-worth.blogspot.com/2016/04/a-guide-to-naming-variables.html>  and mmm, waffles: <http://journal.stuffwithstuff.com/2016/06/16/long-names-are-long/>

**Avoid magic numbers or unnamed numerical constants**. Look at this example - what's wrong?

# Passwords have to be at least 7 characters

if len(password) < 7:

   print('your password is too short')

What does 7 mean? Letters? Penguins? Waffles? We can figure it out by looking at the code, but code is hard to understand, and adding more overhead to reading and understanding is not desirable. This is better,

MIN\_PASSWORD\_LENGTH = 7

if len(password) < MIN\_PASSWORD\_LENGTH:

  print('your password is too short')

This approach also makes it easier to re-use the value somewhere else. And if you use the same value in multiple places in your code, and you need to change it, you can simply change the value of the variable; which helps avoid bugs just in case you didn't change it in all the places. And you can search your code for this variable name. Lots of advantages!

More examples and discussion: <https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants>  <http://stackoverflow.com/questions/47882/what-is-a-magic-number-and-why-is-it-bad>

**Self-documenting code**: avoiding magic numbers and descriptive names, and focusing on clearly expressing your logic, all help your code explain itself. You'll always need some comments, but you can reduce the number of comments you need to write if you can make your code easy to follow. The example above, by replacing the magic number with a variable, becomes more self-documenting.

**Following a code style** for organizing, indenting, commenting, and other code style matters, e.g. Python's pep8 https://www.python.org/dev/peps/pep-0008/. The pep8 package will check your code for some of recommendations <https://pypi.python.org/pypi/pep8>  and some IDEs, such as PyCharm, have this built in.

**Using static analysis tools**, otherwise know as **lint tools** or **linters** e.g. PyLint <https://www.pylint.org/>  check for style, repeated code, unreachable code, making sure class methods have self as the first argument, and many other possible errors. PyCharm runs a linter on your code, or you can install PyLint. Again, other languages have equivalent tools.

**Participate in code reviews**. Seek feedback from peers - other students, and colleagues, as well as your instructors, managers/bosses. What are you doing well, and what can you improve on?

**Tests!!!!** manual and automatic tests (unit tests) very important to verify correctness, and avoid regressions. A regression is where you change something, and the change breaks something else.

**Modular** Each function should do one thing. This makes testing much easier. Group related code into modules. Create a new module if one module is doing multiple things. For example, if you have to do some complicated data processing before returning a response in your web app, create a new module and put the processing code there, not in a route handler.

**Clarity is better than cleverness** being clever with code, trying to save space or writing as few lines as possible, is typically not recommended. See the very first line of this file! Making your code easy to follow and understand is better than saving some space in your code files. Space in the repository is cheap, but bugs and code maintainers are expensive.

**Logging. not print()** Unless your program is a command line app, and it needs to print messages to the user, get rid of print statements. Your code should avoid print statements that report on what's happening in the program, or are used for printing exceptions, or for debugging. This also applies to Console.Write() and console.log() and System.out.println() or equivalent. Debugging info, and messages about program state or errors, should be sent to logs, not the console/terminal. A logging system can be used to store logs, search for logs of particular severity, automatically notify appropriate people if a particular error happens... print() can't do any of that. Python uses <https://docs.python.org/3/library/logging.html>; Django has <https://docs.djangoproject.com/en/1.11/topics/logging/> and other languages also have logging packages.

**Extensibility** If your program has to deal with 4 of something, are those exact 4 things hard-coded? How difficult would it be to add a 5th thing? Or replace one of the things? Because extra features are always requested, and that 5th thing is probably going to be a requirement one day.  Example: building a card game app that deals with exactly two players - how hard would it be to add support for 3 or more players?

**Performant** this is at the end for a reason. Make your code work. Focus on simplicity, correctness, and clarity, as above. Test it, and make sure it does work. And then, and only if it's too slow, make it faster. If, you even need to. Computers are pretty speedy, and most code performance issues in modern apps come from network requests, database queries, file IO, and other external factors.  Code profilers like Python's built in <https://docs.python.org/3.6/library/profile.html> can help determine which functions/methods your code is spending the most amount of time in, and whether it's something you may be able to speed up by re-writing code, or if it's an external issue.