#11: Writing Good Code

SAMS PROGRAMMING C

Course Wrap-Up Thoughts

Tetris Status Report

Quiz Tomorrow (office hours tonight!)

No Lab on Thursday

Letters of Recommendation

Review from Last Week

Create interactive programs with **input/output streams**

Create interactive graphic programs with **keyboard and mouse events**

Create interactive graphic programs with **time-based animations**

Today's Learning Goals

Recognize several different **metrics** for measuring code quality, including:

- How **correctness** can be measured at various levels
- Why **style** makes a difference in code quality
- How **efficiency** can be measured
- Which **high-level metrics** are worth considering in large projects

Metrics of Software

So far, we've written code with only one goal: the code should pass the test cases. This doesn't take into account many other features of the code that are important!

We're going to discuss multiple **ways of evaluating code**, metrics that you'll need to consider when building applications for broader use.

We won't delve deeply into any of these topics; instead, you should consider this a survey of different topics that can help you identify core areas of computer science you might be interested in.

Correctness

Correctness

First and foremost, when we write code, we want it to **work correctly**. If our code doesn't work as expected, it's not useful at all!

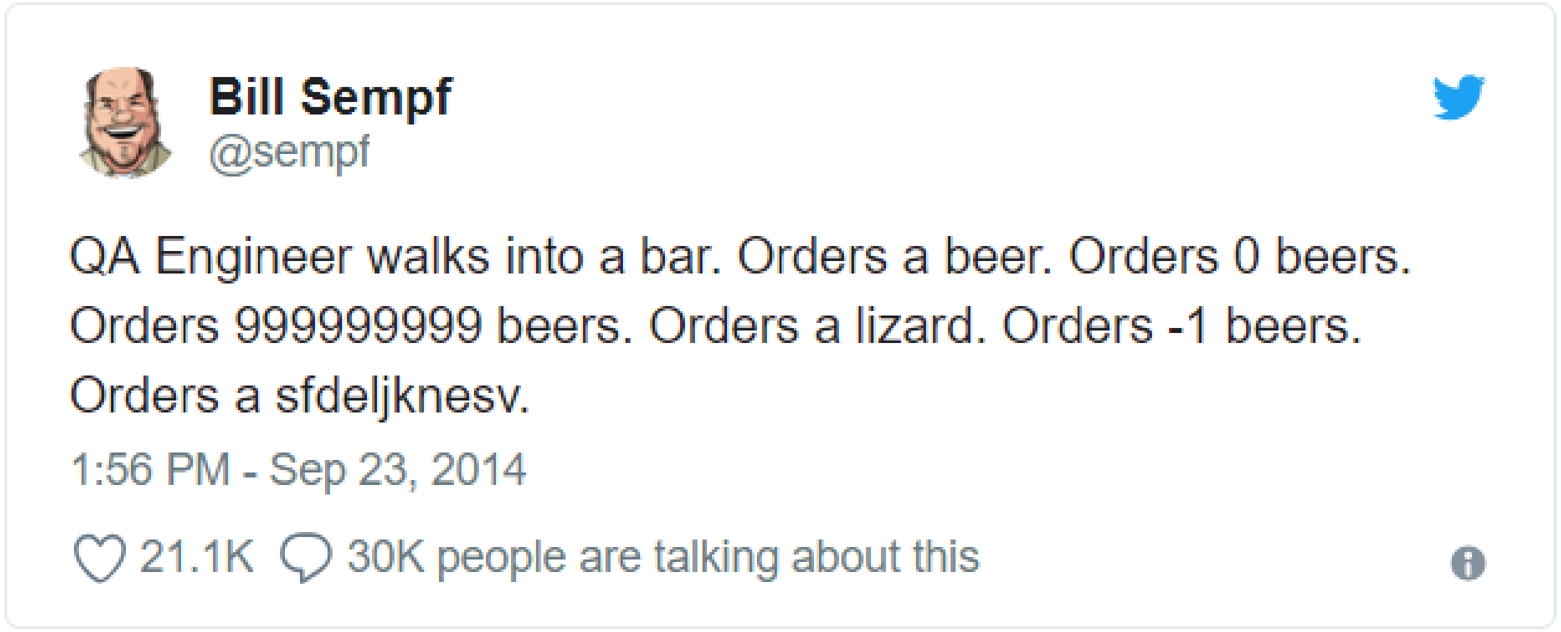
There are multiple ways of thinking about program correctness:

- Low bar: does it pass a well-designed set of test cases?
- Medium bar: does the code maintain certain contracts when run on input?
- High bar: can we mathematically prove that it matches expectations?

The more important your code is, the higher you'll need to set the bar for correctness.

Testing as a Job

Some computer scientists do nothing but testing as their jobs!


Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

1:56 PM - Sep 23, 2014

21.1K 30K people are talking about this

Designing Good Test Cases

Remember the lessons learned in Week 2: you need to cover a variety of possible inputs, including:

- normal cases
- large cases
- edge cases
- special cases
- varying results

When testing interactive applications, mess around with the app and try doing unexpected things to make sure the code handles all inputs.

Code Contracts

Test cases measure a program's correctness based on whether its output matches the expected output on a certain input. But they don't tell us anything about code's behavior **during** program execution!

A code contract is a set of **preconditions**, **postconditions**, and **invariants** that must be true whenever the code is run. These conditions let us ensure certain properties of the code.

- Preconditions are restrictions on the inputted values- they must be a certain type or a certain size.
- Postconditions are restrictions on the returned value- it must have a certain property.
- Invariants are restrictions that must be maintained as the code is run- perhaps a value must always be positive, or always a certain length.

If a contract is violated, an error is thrown so that the programmer can immediately notice the problem and fix it.

Proving Correctness

Code contracts help us ensure certain properties as code is being run, but they are only as good as the tests we run. If we want to ensure that code will **always** work, we have two options: test the function on every possible input, or prove that it works mathematically.

Testing on every possible input is only possible in a subset of cases, when we know all the possible values that can be inputted. This is the case with Booleans and enumerated types (types that have a finite set of specified values).

Proving a program is correct more often means that you need to prove the **algorithm** is correct. This can be done with several proof methods, including induction. A program can implement a proven algorithm **incorrectly** (by having a bug), so you should always check your work!

Style

Style

Once code is correct, we should consider how that code will be used by other people. This is covered under the idea of **programming style**. A program's style drastically affects how easy it easy to read and debug its code!

Programming style is similar to writing style. Some guidelines are almost universally agreed upon (including at least some comments, avoiding duplication of code); some are much more contentious (variable name casing, tabs vs. spaces). Different languages, classes, and companies have different **style guides** that recommend what the program's style should look like.

Official Python style guide: https://www.python.org/dev/peps/pep-0008/

Clarity

Clarity refers to how easy it is to **read the code and understand what it is supposed to do**. Reading clear code should be as easy as reading natural text.

- Ideate **well-designed algorithms** that match the problem statement
  - Avoid overly complex logic- you can usually simplify an algorithm into a set of meaningful steps
- Use **meaningful variable names** that explain the variable's purpose
  - Exceptions can be made for common standards: i and j for indices, x and y for numbers, s for string
- Write **clear comments** to explain any piece of code that is not immediately obvious
  - Caveat: you can write too few comments, but you can also write too many! Don't over-explain code!
- Avoid **overly long lines of code**
  - General rule: no line of code should be longer than 80 characters. Break the code up into multiple lines instead.
- Remove **any unnecessary code** once you're done
  - This includes debugging code, unused variables, and commented-out code.

Maintainability

Maintainability refers to how easy it is to **maintain and update code over time**. Maintainability is especially important when writing code as a team or code that will be made open-source.

- Write a comment for **every function in the code**
  - The comment should briefly explain what the function does, what input it expects, and what the output will be.
- Write comments to explain **non-obvious decisions** in the code
  - If complex code can't be simplified for a specific reason, state that reason!
- Follow **style guidelines** when working with a team
  - If everyone in your team uses underscore_variable_naming, you should too.
- **Never copy and paste code!!!**
  - If you find yourself copying and pasting code, put it in a helper function instead.

Generality

Write code that works on **generalized inputs and situations**, rather than setting restrictions for what the input must be.

- Avoid using **magic numbers** (numbers hard-coded into expressions)
  - Exception: this is okay if the number's purpose is obvious (for example, checking evenness with x % 2).
  - Put all other numbers into well-named variables instead.
- Use **helper functions** where appropriate to organize functionality.
  - In general, if a program is 20+ lines long, it's trying to do too much. Separate the steps out into helper functions and call them instead.
- Handle **unexpected input** at the beginning of the function.
  - Check for incorrect types, empty strings/lists, incorrectly-formatted files, etc.

Efficiency

Program Runtimes

So far, we've written programs that can run pretty quickly, usually because we're testing them on fairly small inputs.

When we test on larger inputs, programs can take longer to run...

Example: isLegalSudoku

```
Removed for online version.
```

Algorithmic Efficiency

These two implementations have one core difference: **the number of operations that need to be performed.**

If we say `N = len(board)`, then the left `isLegalSudoku` calls each `isLegal` function `N**3` times. The right `isLegalSudoku` only calls each `isLegal` function N times!

In general, to improve algorithm runtime (and efficiency), we want to **reduce the number of operations we need to perform**. We especially want to pay attention to operations that **grow larger as the input grows larger**- that is, we care about operations that depend on N!

Calculating Runtime

We can actually test how long a program takes to run using the **time** library:

https://docs.python.org/3/library/time.html

`time.time()` returns the number of seconds since the epoch (when computers start recording time). We can use it to measure how long it takes a function to run:

```
t1 = time.time()
runFunction()
t2 = time.time()
print("Time passed: " + str(t2 - t1))
```

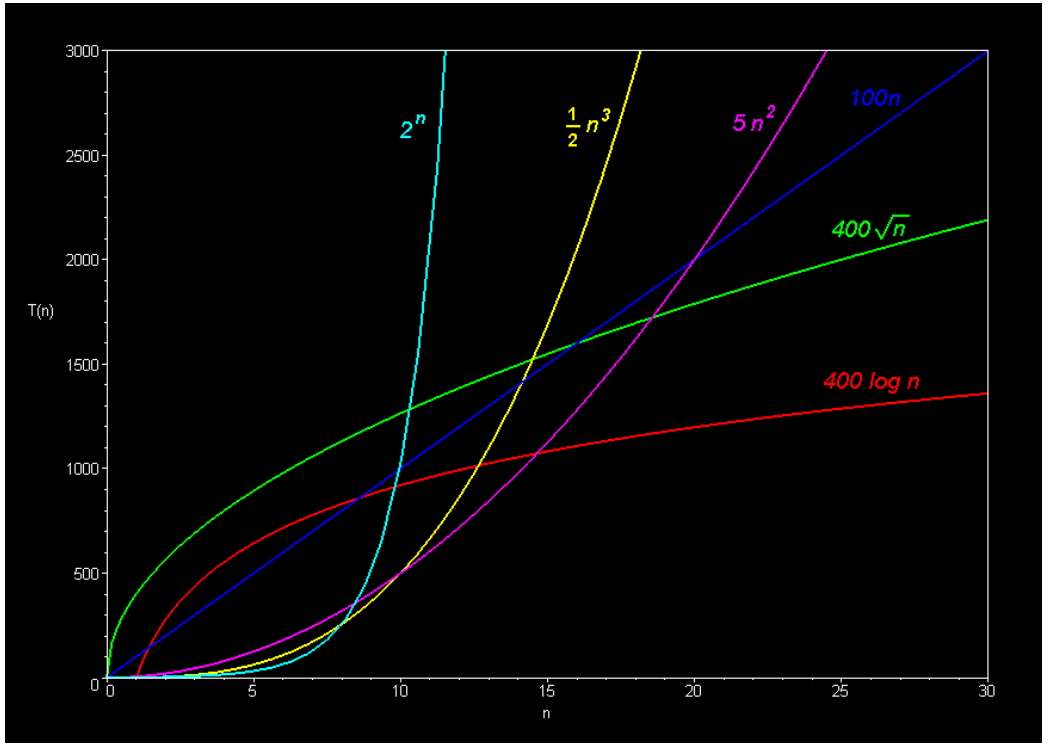
Big-O Notation and Function Families

Computer scientists use **Big-O notation** to mathematically approximate the runtime of a program. Big-O notation tells us the **function family** a program runs in assuming the **worst possible case**.

Constant time (`O(1)`) and linear time (`O(N)`) are both fairly fast.

Polynomial time (`O(`N^2`)`, `O(`N^3`)`) is slow.

Exponential time (`O(`2^N`)`, `O(N!)`) is horrible!



Improving Efficiency

Making code run fast is important if we want other people to actually use our code! No one wants to use an app that takes a full minute to load.

We can use tools like **performance profilers** to help us understand where our code is being slowed down, so we can figure out what needs to be fixed.

Python built-in profiler: https://docs.python.org/3/library/profile.html

Sometimes, though, code won't work unless we use an exponential-time algorithm. It's important to recognize when slow algorithms are necessary, and when they can be improved.

High-Level Metrics

High-Level Metrics

As you start developing code at a higher level, it's important to consider not just how the code works, but how people will interact with it. This can cover a large number of possible metrics, including:

- **Usability:** how easy is it for a user to interact with the program?
- **Accessibility:** does the program work properly for differently-abled people?
- **Security:** does the program keep 'bad actors' from accessing/using it improperly?
- **Privacy:** does the program protect any user data it gathers?

Usability

When designing for usability, we want to ensure that the program is **natural and easy to use**, whether by an expert or a brand new user.

To do this, you must determine **how users use your program** and **what their expectations are**. Good design matches user expectations as closely as possible, so that the process of using an application feels simple.

To make a system more usable, it is important to **test it with real users!** Observing how people interact with a system can teach you a great deal about how to improve it.

Accessibility

When designing for accessibility, we want to make sure that the program is **usable for all possible populations**. This includes:

- people who have a visual or hearing impairment
- people who are colorblind
- people who primarily speak a different language
- people who have trouble manipulating traditional keyboards and mice
- people who are not used to using computational devices

There are several frameworks that provide guidance on accessibility, including how to make applications work with screen readers and different input devices. Here's an example of a standard for the internet: https://www.w3.org/standards/webdesign/accessibility

Security

When designing for security, we need to **guard against potential adversaries** who may want to gain access to a system to steal information, manipulate data, or take down a system.

Most security breaches happen due to **weak links** in a chain of communication. For example:

- If an application makes locals process data visible and editable publicly, an adversary can modify it.
- If an application does not encrypt communication to a server, an adversary can intercept and read it, and send fake messages
- If a programmer's login information is compromised, an adversary can log directly into the system!

Some security breaches happen due to **poorly guarded input**- it's important to safeguard any input that the user enters, to make sure it doesn't cause errors which can lead to bad behavior.

Privacy

When designing for privacy, we need to **ensure that user data is protected and only visible as the user decides**. Privacy and security are often considered together, as maintaining privacy often involves securing a system's data against adversaries.

Privacy is also important to consider when designing a system's core features. **What data is considered open, and what is considered sensitive?** This varies across different users, so allowing for user control is important. When determining how to monetize a system, privacy is also a crucial concern- what is acceptable to share with advertisers?

Privacy has recently become a legal matter as well, with the EU's **General Data Protection Regulation (GDPR)** requiring that companies approach data privacy in new ways that ensure greater user control over data. The question of whether privacy is a core human right will likely be hotly debated during our lifetime.

Today's Learning Goals

Recognize several different **metrics** for measuring code quality, including:

- How **correctness** can be measured at various levels
- Why **style** makes a difference in code quality
- How **efficiency** can be measured
- Which **high-level metrics** are worth considering in large projects