CSC290 Communication Skills for Computer Scientists

Lisa Zhang

Lecture 9; Nov 12, 2018

Today

1. Writing Clean Code
2. Writing Commit Messages
3. Reviewing Code

Also:

- Critical Review 2

Writing Clean Code

What does this code do?

```
define f(x): # x = "20181112"
    y, m, d = x[:4], x[4:6], x[6:8]

    if m == "01": m = "Jan"
    if m == "02": m = "Feb"
    if m == "04": m = "Apr"
    if m == "05": m = "May"
    # ... etc ...

    return int(y), m, int(d)
```

Agree or disagree?

“Programs are meant to be read by humans and only incidentally for computers to execute.”

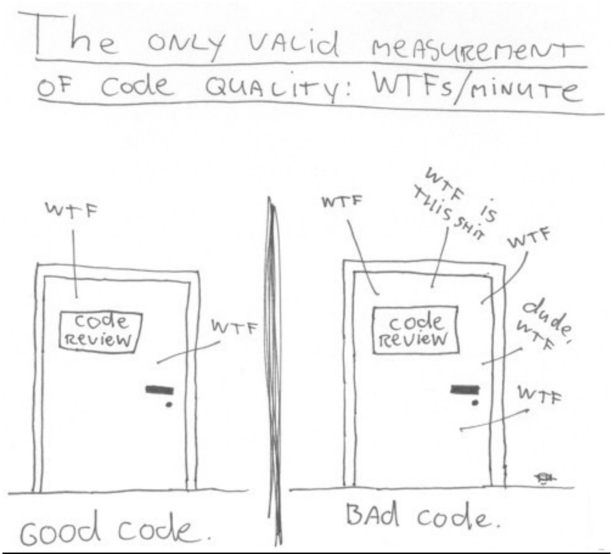
– Donald Knuth

What is good code?

What is good code?

- Follow the appropriate **conventions**
- Clear variable names
- Well documented (or, speaks for itself)

Good code



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Following appropriate conventions

Different organizations will have different conventions.

- https://google.github.io/styleguide/javaguide.html
- https://github.com/google/styleguide/blob/gh-pages/pyguide.md

Larger organizations will have more formal conventions.

- Use tools to automatically check whether your code follow conventions

Python Convention Examples:

- No trailing white spaces
- Four-space tabs (instead of two-space tabs, and tab characters)

Python Convention Examples:

- No trailing white spaces
- Four-space tabs (instead of two-space tabs, and tab characters)
- Indentation

```
def draw_box(self, width, height, border,
             color='black', highlight='none'):
```

Python Convention Examples:

- No trailing white spaces
- Four-space tabs (instead of two-space tabs, and tab characters)
- Indentation

```python
def draw_box(self, width, height, border,
             color='black', highlight='none'):
```

- Avoiding complex list comprehensions:

```python
result = [(x, y)
          for x in range(10)
          for y in range(5)
          if x * y > 10]
```

Abstract over structures

```python
def month_name(month):
    if month == 1:
        return "January"
    if month == 2:
        return "February"
    if month == 3:
        return "March"
    ...
```

Abstract over structures

```python
MONTH_NAME = ["January", "February", "March", ...]

def month_name(month):
    return MONTH_NAME[month]
```

Reduce code repetition

```python
def make_egg():
    egg = take_out("egg")
    cooked_egg = cook(egg)
    plated_egg = plate(cooked_egg)
    return plated_egg

def make_ham():
    ham = take_out("ham")
    cooked_ham = cook(ham)
    plated_ham = plate(cooked_ham)
    return plated_ham
```

Don't rewrite the builtins

```python
def round(num):
    frac = num % 1
    if frac >= 0.5:
        return (num - frac + 1)
    return (num - frac)
```

Don't re-write code that other people in your project have already written.

Reduce nesting (exit early)

```
def foo(n):
    if n > 0:
        do_work_a(n)
        do_work_b(n)
        ...
```

versus:

```
def foo(n):
    if n <= 0:
        return
    do_work_a(n)
    do_work_b(n)
    ...
```

Writing Testable Code

- **Unit test** verifies the behaviour of a small part of your code
  - Easy to write and run
- **Integration test** verifies that components interacts well with each other
  - Difficult to write and run

So what makes code easier to test?

Writing Testable Code

- **Unit test** verifies the behaviour of a small part of your code
  - Easy to write and run
- **Integration test** verifies that components interacts well with each other
  - Difficult to write and run

So what makes code easier to test?

- Each function should do one thing only.
- Isolate functions that interact with external systems (file system, database)
- Prefer **pure** functions
  - Function whose output is deterministic given its arguments

Code that is difficult to unit-test

```python
def read_file_and_compute_total(file):
    total = 0
    for line in open(file):
        item, price = line.split(",")
        price = float(price)
        if item not in FOOD_LIST:
            total += price * 1.13
        else:
            total += price
    return total
```

Code that is easier to unit-test

```python
def read_product_price(file):
    products = []
    for line in open(file):
        item, price = line.split(",")
        products.append(item, float(price))
    return products

def compute_total(item, price):
    if item in FOOD_LIST:
        return price
    return price * 1.13

def read_file_and_compute_total(file):
    return sum([compute_total(item, price)
                for (item, price)
                in read_product_price(file)])
```

Variable Naming

*There are only two hard things in Computer Science:
cache invalidation and naming things.*

– Phil Karlton

Most Important Naming Consideration

- Does the name fully and accurately describe what the variable represents?
- Name should have the right level of specificity
    - The larger the scope, the more specific the name
    - Reserve single characters names for short loops only
    - Use `i`, `j`, `k` for integer loop indicies (why not `l`?)
- Name should be easy to **search** for

Avoid common, meaningless names

- flag
- status
- data
- variable
- tmp
- foo, bar, etc.

Comments

- Comments should **explain why** the code is what it is.
- Comments should never repeat the code.
- Ideally, the code will make sense without any comments.

Comments that repeat the code:

```python
def compute_total(item, price):
    if item in FOOD_LIST: # item is a food
        return price       # just return the price
    return price * 1.13    # multiply price by 1.13
```

Comments that explain the code

```
def compute_total(item, price):
    if item in FOOD_LIST:
        # food items are not taxed
        return price
    return price * 1.13 # tax rate is 13%
```

Comments that mark the code

```
def compute_total(item, price):
    if item in FOOD_LIST:
        return price
    return price * 1.13 # TODO: make tax rate a param
```

. . . but clean these up, ideally before committing.

Other comments:

- Block comments to lay out code
- Comments that describe the code's intent
- Comments that summarizes a chunk of code
- Information like copyright notices, references, etc.

Exercise: rewrite this code

```
define f(x): # x = "20181112"
    y, m, d = x[:4], x[4:6], x[6:8]

    if m == "01": m = "Jan"
    if m == "02": m = "Feb"
    if m == "04": m = "Apr"
    if m == "05": m = "May"
    # ... etc ...

    return int(y), m, int(d)
```

Critical Review

Second Critical Review

The text is a little more difficult

Revisiting Why Students Drop CS1. Petersen et al. 2016.

https://dl-acm-org.myaccess.library.utoronto.ca/citation.cfm?id=2999552

Start by reading the paper on how to read papers.

https://blizzard.cs.uwaterloo.ca/keshav/home/Papers/data/07/paper-reading.pdf

Qualitative Study

- This is a **qualitative** study (as opposed to **quantitative**).
  - Authors conduct interviews, and analyze the transcript.
  - No statistics.
- Conducting interviews is more time-consuming than conducting surveys, so the sample size is usually smaller.

Understanding

- You might not understand everything about the paper, that's okay.
  - Don't worry about section 3.2 (grounded theory).
  - Do not read the related work material.
- You should be able to understand the methodology and the conclusion.

Critical Review Tone

- You might know some of the authors of this paper. Consider that they may read your review.
- Write in a professional tone.
- “Critical” does not mean “criticize”: Talk about both the positives and the negatives of the paper
- Focus on the issues:
  - Is the data appropriate?
  - Is the methodology appropriate?
  - What could we have done to make the research even better?

How to begin

- Read the abstract
- Read the headings
- Read the introduction & conclusion
- Read anything sections that stand out to you
- Read the entire article
- Summarize the article in your own words
- Think about your review
- Write your review

Critical Review Format and Submission

- Submit a **PDF** file on **MarkUs**
- I strongly recommend that you use LaTeX
  - Used by the computer science and math communities
  - LaTeX will format your citations for you
  - Starter code: https://v1.overleaf.com/read/pxhszkghmkvf
- Submission through MarkUs

Commits and Commit Messages

What is a “commit”?

- Small set of modifications to a code base
    - Each commit should contain one (atomic) change
    - Commits should be standalone (independent of other commits)

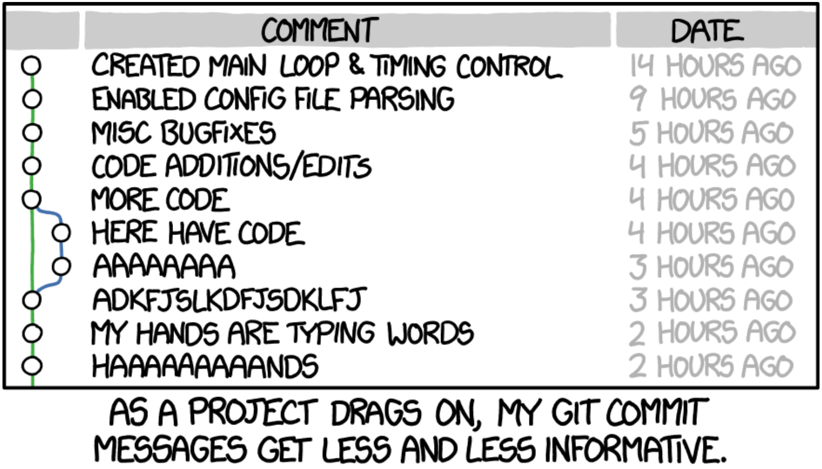
Open Source Examples

- Chromium
  - https://github.com/chromium/chromium/commits/master
- NumPy
  - https://github.com/numpy/numpy/commits/master
- Evennia (python text-based game library)
  - https://github.com/evennia/evennia/commits/master

Commits

- Do not fold small changes (e.g. typo fixes) into another commit
- When commits are atomic and standlone, they can be applied and reverted independently
- The **commit message** summarizes the code changes
- Makes the version control utilities much easier to use

Commit Messages



Guidelines

- Commit message guidelines differ by company.
- Bigger organizations programmatically read/modify commit messages.
  - Chromium's commit messages have a lot of boilerplating.

Common guidelines

From https://chris.beams.io/posts/git-commit/

- Separate subject from body with a blank line.
- Limit the subject line to 50 characters.
- Capitalize the subject line.
- Do not end the subject line with a period.
- Use the imperative mood in the subject line.
- Wrap the body at 72 characters.
- Use the body to explain what and why vs. how.

Example:

You add the following lines to a file called “tictactoe.py”

```
+ # Python TicTacToe game on the command line
+ # Author: Mr. Pirate <mr@pirate.com>
```

What should your commit message be?

Example:

What is wrong with the following commit messages?

```
Added comments to tictactoe
author and file description in tictactoe.py
comment description and author and email to the first few l
Add description, author, and email.
```

Reviewing Code

Code Review

Most companies use “code review” to ensure high code quality.

1. Code writer submits code for review.
2. One or more reviewers (peers) read the code.
3. If reviewers notice areas of improvement, reviewers will request for changes.
4. Code writer works with the reviewer to address any raised issue (back to step 2)
5. When all reviewer concerns are addressed, the code is accepted (pushed).

Why code review?

- Encourage committers to write clean code.
- Share knowledge across team members.
- Encourages consistency in the code base.
- Help prevent bugs and other issues.

In most large organizations, **all** code, no matter who wrote it or how large/small it is, need to be reviewed.

What does the reviewer do?

- Does the code accomplish the author's purpose?
- What is the author's approach? Would you have solved the problem differently?
- Do you see potential for useful abstractions?
- Do you spot any bugs or issues?
- Does the change follow standard patterns?
- Is the code easy to read?
- Is this code documented and tested?

Example:

- https://github.com/evennia/evennia/pull/1666
- https://github.com/numpy/numpy/pull/11721
- https://github.com/numpy/numpy/pull/10931
- https://github.com/numpy/numpy/pull/10771

How to Review Code

- Critique the code, not the author.
  - "*your* code has a bug" vs "the code has a bug".
- Ask questions (perception checking!).
- Reviews should be concise and actionable:
  - Make it clear what you are asking for
- Don't be mean.

Responding to Code Review

- Be civil, and be open minded.
- First time you submit code, you will have *many* comments, don't feel daunted.