15-122: Principles of Imperative Computation

Style Guide

Nivedita Chopra

0. Making your code look pretty

You want your code to look nice. Not only so that your TA enjoys grading it, but also for yourself, when you look back on your code later, long after completing the assignment.

Here's how to make the document itself look nice:

- **Indentation** - Though not required in C0 and C (as it is in Python), proper indentation makes code look nice. You should indent every time you open a block of code like an `if` statement or a `for` or `while` loop.

  Avoid using tabs to indent your code. They render differently on different systems (and especially badly on Autolab). Instead use *either* 2 *or* 4 spaces, depending on your preferences. Choose one and stick to it! You should set your tabs to 2 or 4 spaces in your .emacs or .vimrc file.

- **Whitespace** - Don't make your code terribly cramped up so that it's hard to read. Use blank lines to separate unrelated blocks of code and to give your program a structure. Within a line use spaces to to separate characters but stay consistent.

  Use the same amount of space on either side of an infix operator (either 0 or 1 spaces), *do* `2+3` or `2 + 3`, *not* `2+ 3`. Use spaces around parentheses: `(2 + 3) * (4 + 5)` *is better than* `(2 + 3)*(4 + 5)`. Avoid trailing whitespace at the end of lines.

- **Line Length** - Limit your lines of code to a *maximum of 80 characters* in length. This is a very well-known convention. Higher level CS classes enforce it very strictly, so it is in your best interest to follow this strictly in this class as well. You can use the unix command `wc -L mycode.c0` to see the max line length in a file mycode.c0

  After submitting your code, always open up your handin on Autolab to make sure it looks all right. There may be some subtle formatting errors in your code, which will become pretty obvious on Autolab.

1. Making your code easier to understand

Okay, so you have pretty-looking code because you followed the steps above. But what does this pretty-looking code do? You want to make sure that other people (and yourself, at a later date) are able to understand your approach to a problem by looking at your code for solving it.

Here's how to make your code easier to understand:

- **Commenting** - Comment your code well, so that a person glancing at it can understand most of what you're trying to do. This means you shouldn't be writing large paragraphs as comments, but rather a couple of lines to explain the purpose of a helper function or a tricky bit of code.

  To decide when to explain what you're doing - think of it as: "Was I able to come up with this easily? And now that I've come up with it, can I explain it after a week?" If the answer to both of these questions is "No", definitely comment your code (but feel free to comment in other cases too!)

- **Dead Code** - Dead code is anything that does not contribute to your solution to the task at hand. This includes `print` statements for debugging and code that is never executed when your program is run.

  To prevent confusion to a reader, completely *delete* all dead code. Simply commenting it out is bad style, as it reduces the value of your more useful comments.

- **Meaningful Names** - You should provide meaningful names for all your variables and functions. Avoid generic variable names like `i, j, x, x1, x'`. Restrict them to variables like the counters within loops, etc. Name variables in a manner that relates to the value that is stored in them during the program and functions in a manner that reflects the task that they perform.

  To name variables with multiple words, use *either* underscores (`foo_bar_foo`) *or* camel case (`fooBarFoo`). Don't mix them up like `foo_barFoo` and once you pick one style, follow it for all names throughout you program.

- **Use Helper Functions** - If you find yourself writing similar code with a few changes at multiple places in your program, consider putting that block of code into a helper function and calling it in the function that you are writing. This makes it easier for someone to read your helper function and understand what it's doing, independently of the remaining program. Always include a comment just above a helper function describing its purpose.

2. Making your logic clearer

Sometimes, things that you do in a convoluted manner can be expressed much more easily in a simple statement. Here are some examples:

Rather than	Simply do

`if(condition){`
`  return true;`
`} else {`
`  return false;`
`}`
	`return condition;`

`if(condition){`
`} else {`
`  do_something();`
`}`
	`if(!condition){`
	`  do_something();`
	`}`

`if (some_boolean_statement == true){`
`  do_something();`
`}`
	`if (some_boolean_statement){`
	`  do_something();`
	`}`

`if (condition_1){`
`  if(condition_2){`
`  do_something();`
`  }`
`}`
	`if (condition_1 && condition_2){`
	`  do_something();`
	`}`

`if (i >= 0 && i <= n){`
`  do_something();`
`}`
	`if (0 <= i && i <= n){`
	`  do_something();`
	`}`

And always remember:

"Programs must be written for people to read, and only incidentally for machines to execute."
H. Abelson and G. Sussman (in 'The Structure and Interpretation of Computer Programs')