Computer Science @ The University of Waterloo

CS 115, CS 116, CS 135, & CS 136 Common Style Guide

Last Updated: 2019.12.12

1 Introduction

The code you submit for assignments, as with all code you write, can be made more readable and useful by paying attention to style. This includes the placement of comments, whitespace, indentation, and choice of variable and function names. None of these things affect the execution of a program, but they affect its readability and extensibility. As in writing English prose, the goal is communication, and you need to think of the needs of the reader. This is especially important when the reader is assigning you a grade. This guide is meant to serve as a language independent common formatting guide for all of your programs that you will write in first year computer science courses.

1.1 A Warning on Examples From Other Sources

The examples in the presentation slides, handouts and tutorials/labs are often condensed to fit them into a few lines; you should not imitate their condensed style for assignments, because you do not have the same space restrictions.

For your assignments, use the coding style in this guide, not the style from another textbook, or the condensed style from the presentation slides.

1.2 Warning About Copying and Pasting

Copying and pasting can make your assignment unmarkable. It may lead to characters appearing in code that our auto graders cannot read correctly and hence will be marked as incorrect. Do not include anything other than ASCII characters (characters found on a standard US keyboard) into your text files.

2 Assignment Formatting

2.1 Comments

There are three types of comments: comments on a full line (use `;;` in Racket, `##` in Python or `//` in C), those used in-line (use `;` in Racket, `#` in Python or `//` in C) or block comments (use `/* comments */` in C). Comments should be used for documentation purposes and to explain *why* code does what it does.
Use in-line comments sparingly. If you are using standard design recipes and templates, and following the rest of the guidelines here, you should not need many additional comments. Any such comment can either be put on its own line, or tacked onto the end of a line of code, providing it fits.

2.2 Header

Your file should start with a header to identify yourself, the term, the assignment and the problem. There is no specifically required format, but it should be clear and assist the reader. The following is a good example in Racket.

```
;;
;; ***************************************************
;;    Rick Sanchez (12345678)
;;    CS 135 Fall 2019
;;    Assignment 03, Problem 4
;; ***************************************************
;;
```

where you should replace `;;` with `##` in Python and by `//` in C.

2.3 Whitespace, Indentation and Layout

Whitespace should be used to make it easier to read code. This means not having too much or too little whitespace in your code. In what follows, we give an example of what our whitespacing will look like in our model solutions though you can deviate from this slightly as long as the code is still readable.

- Insert two consecutive blank lines between functions or “function blocks”, which include the documentation (*e.g.,* design recipe) if applicable for the function.
- Insert one blank line before the function header and one after the function body.

- Blank lines may appear in function blocks but do not use more than one consecutive blank line within a function block.

If the question asks you to write more than one function, the file should contain them in the order specified by the assignment. Helper functions are placed either all above the assignment function(s) they are helping or underneath the functions they are helping. Remember that the goal is to make it easier for the reader to determine where your functions are located.
Indentation plays a big part in readability. Some languages like Python force indentation while others like Racket and C do not. Read the section specific style guide to determine how to properly indent your code in your respective language. When to start a new line (hit enter or return) is a matter of judgment. Try not to let your lines get longer than about 70 characters, and definitely no longer than 80 characters. You do not want your code to look too horizontal, or too vertical.

Style marks may be deducted if you exceed 80 characters on any line of your assignment.

2.4 Constant (Variable) and Function Identifiers

2.4.1 Naming Functions, Parameters, Variables and Constants

Function and parameter names should be meaningful, but not awkwardly long nor cryptically short. The first priority should be to choose a meaningful name. Names like `salary` or `remittance` would be appropriate in a program that calculates taxes. Sometimes a function will consume values that don't have a meaning attached, for example, a function that calculates the maximum of two numbers. In that case, chose names that reflect the structure of the data. That is, `n` is for numbers, `i` for integers, `lst` for a list or `lon` for a list of numbers, and so on. Names that are proper nouns like Newton should always be capitalized, Otherwise, use the following function naming conventions:

- In Racket, use lower-case letters and hyphens, eg. `top-bracket-amount`
- In Python and C, you may use one of the following:
  - Snake Case: use only lower case letters and separate words with an underscore such as `top_bracket_amount`.
  - Camel Case: begin with a lower/upper case letter; following words are capitalized. For example `topBracketAmount` or if capitalizing the first word, `TopBracketAmount`.

It is important to pick one convention and to be consistent with your naming convention in any given file. In some cases, this will mean following the convention given by the assignment (for example, if the function you are to write is `my_fun`, then your helper functions should also use snake case.

2.4.2 Constants

Constants should be used to improve your code in the following ways:

- To improve the readability of your code by avoiding "magic" numbers. For example, if you have code dealing with tax rates like Ontario's HST, you might want a constant such as `taxes` or `hst` and have this value set to 0.13.
- To improve flexibility and allow easier updating of special values. If the value of `taxes` changes, it is much easier to make one change to the definition of the constant than to search through an entire program for the value `0.13`. When this value is found, it may not be obvious if it refers to the tax rate, or whether it serves a different purpose and should not be changed.
- To define values for testing and examples. As values used in testing and examples become more complicated (*e.g.,* lists, structures, lists of structures), it can be very helpful to define named constants to be used in multiple tests and examples.

2.5 Summary

- Use two comment symbols for full-line comments and use one comment symbol for in-line comments, and use them sparingly inside the body of functions.
- Provide a file header for your assignments.
- Make it clear where function blocks begin and end
- Order your functions appropriately.
- Avoid overly horizontal or vertical code layout.
- Use reasonable line lengths.
- Choose meaningful identifier names and follow our naming conventions.
- Avoid use of "magic numbers".

Style marks may be deducted if you have poor headers, identifier names, whitespace, indentation or layout.

3 The Design Recipe: Functions

Warning! This style guide will be used for assessment (*i.e.,* assignments and exams).

We hope you will use the design recipe as part of the process of working out your solutions to assignment questions. If you hand in only code, even if it works perfectly, you will earn only a fraction of the marks available. Elements of the design recipe help us to understand your code.

3.1 Purpose

The purpose statement has two parts: an illustration of how the function is applied, and a brief description of what the function does. The description does not have to explain how the computation is done; the code itself addresses that question.

- The purpose starts with an example of how the function is applied, which uses the same parameter names used in the function header.
- Do not write the word "purpose".
- The description must include the names of the parameters in the purpose to make it clear what they mean and how they relate to what the function does (choosing meaningful parameter names helps also). Do not include parameter types and requirements in your purpose statement — the contract already contains that information.
- If the description requires more than one line, "indent" the next line of the purpose 2 or 3 spaces.
- If you find the purpose of one of your helper functions is too long or too complicated, you might want to reconsider your approach by using a different helper function or perhaps using more than one helper function.

3.2 Contract

The contract contains the name of the function, the types of the arguments it consumes, and the type of the value it produces. The contract is analogous to functions defined mathematically that map from a domain to a co-domain (or more loosely to the range of the function).

```
;; quot: Nat Nat -> Nat
```

See your specific course style guide for information about valid types in the language being used. What follows are some of these more common types:

`Any`	Any value is acceptable
`(anyof T1 T2...)`	Mixed data types. For example, `(anyof Int Str)` can be either an `Int` or a `Str`; `(anyof Int False)` can be either an `Int` or the value `False`, but not `True`.
`Int`	Integers: `...-2, -1, 0, 1, 2...`
`Nat`	Natural Numbers (non-negative Integers): `0, 1, 2...`
`Num` or `Float`	Any non-integer value
`Str`	String (*e.g.,*`"Hello There"`, `"a string"`)
`X, Y, ...`	Matching types to indicate parameters must be of the same type. For example, in the following contract, the `X` can be any type, but all of the `X`'s must be the same type: `my-fn:  X (listof X) -> X`
`Bool`	Boolean values (`True` and `False`)
`(listof T)`	A list of **arbitrary length** with elements of type T, where T can be any valid type. For example: `(listof Any)`, `(listof Int)`, `(listof (anyof Int Str))`.
`(list T1 T2...)`	A list of **fixed length** with elements of type T1, T2, etc. For example: `(list Int Str)` always has two elements: an `Int` (first) and a `Str` (second).
`User_Defined`	For structures/new class user-defined types. Capitalize your user-defined types.

Please note that `Num` is the type used in Racket (CS115/135) but `Float` is the type used in Python and C (CS 116/136).

3.2.1 Additional Contract Requirements

If there are important constraints on the parameters that are not fully described in the contract, add an additional **requires** section after the contract. Single requirements can be on one line. Multiple requirements should start on a separate line and each line should be indented 2 or 3 spaces. For example in Racket:

```
;; quot: Nat Nat -> Nat
;; Requires:
;;    n1 >= 0
;;    n2 > 0
(quot n1 n2)
```

Note that in Racket, the function header is after the documentation whereas in Python and C it will come before the documentation.

3.3 Examples

The examples should be chosen to illustrate "typical" uses of the function and to illuminate some of the difficulties to be faced in writing it. Examples should cover each case described in the data definition for the type consumed by the function. The examples do not have to cover all the cases that the code addresses; that is the job of the tests, which are designed after the code is written. It is very useful to write your examples before you start writing your code. These examples will help you to organize your thoughts about what *exactly* you expect your function to do. You might be surprised by how much of a difference this makes.

For recursive data, your examples **must** include *each* base case and at least one recursive case. Examples should cover edge cases whenever possible (for example, empty lists).

3.4 Tests

Make sure that your tests are actually testing every part of the code. For example, if a conditional expression has three possible outcomes, you have tests that check each of the possible outcomes. Furthermore, your tests should be directed: each one should aim at a particular case, or section of code. Some people write tests that use a large amount of data; this is not necessarily the best idea, because if they fail, it is difficult to figure out why. Others write lots of tests, but have several tests in a row that do essentially the same thing. It's not a question of quantity, but of quality. You should design a small, comprehensive test suite.

Never figure out the answers to your tests by running your own code. Work out the correct answers independently (*e.g.,* by hand).

3.4.1 Testing Tips

Parameter type	Consider trying these values
`Num` [or `Float`]	positive, negative, 0, non-integer values, specific boundaries, small and/or large values
`Int`	positive, negative, 0
`Bool`	`true`, `false`
`Str`	empty string (`""`), length 1, length > 1, extra whitespace, different character types, etc.
`(anyof ...)`	values for each possible type
`(listof T)`	empty, length 1, length > 1, duplicate values in list, special situations, etc.
`User-Defined`	special values for each field (structures), and for each possibility (mixed types)

3.5 Additional Design Recipe Considerations

3.5.1 Helper Functions

Do not use the word "helper" in your function name: use a descriptive function name. Depending on which course you are taking, all design recipe elements might not be necessary for helper functions. Purpose and contracts however should always be included. You are not required to provide tests for your helper functions (but often it is a very good idea). In the past, we have seen students avoid writing helper functions because they did not want to provide documentation for them. This is a bad habit that we strongly discourage. Writing good helper functions is an essential skill in software development and having to write a purpose and contract should not discourage you from writing a helper function. Marks may be deducted if you are not using helper functions when appropriate. Helper functions should be placed before the required function(s) in your submission.

Functions we ask you to write require a full design recipe. Helper functions need no examples or tests but must have all of the other design recipe elements as required by your course.

3.5.2 Wrapper Functions

If the required function is the wrapper function, then include the examples and tests with it. Otherwise, follow the same procedure that you would with a helper function above.

3.6 Summary

- Purposes should be brief and describe everything your function is doing.
- Parameters names must be in the purpose and explained.
- Contracts should use appropriate types
- Requirements should be used to explain restrictions on input not able to be captured by a contract.
- Examples for code should cover edge/base cases.
- Tests should be a small comprehensive suite covering both edge and typical cases.
- Helper functions do not need examples and tests but require all other design recipe elements.