**Code and Project documentation framework**

***Draft Guidelines for Code Documentation***

**06/12/2021**

Contents

[Contents 2](#_Toc95479164)

[Introduction – Tīmatanga Kōrero 3](#_Toc95479165)

[Purpose 3](#_Toc95479166)

[Background 3](#_Toc95479167)

[*Why documentation is important:* 3](#_Toc95479168)

[Guidelines for code – Aratohu mā Waehere 4](#_Toc95479169)

[General Principles: 4](#_Toc95479170)

[Requirements of documentation: 4](#_Toc95479171)

[Terminology: 5](#_Toc95479172)

[Conventions: 6](#_Toc95479173)

[Example documentation of specific subcategories of code: 8](#_Toc95479174)

[References – Tohutoro 10](#_Toc95479175)

[Instructions and Author(S) 11](#_Toc95479176)

[Handling Instructions 11](#_Toc95479177)

[Document Production 12](#_Toc95479178)

[Distribution List 12](#_Toc95479179)

Figures

[Figure 1: Line with comment 8](#_Toc95479316)

[Figure 2: Line broken up for readability 8](#_Toc95479317)

[Figure 3: Basic function 8](#_Toc95479318)

[Figure 4: Class Object with functions 8](#_Toc95479319)

[Figure 5: Good and Bad assignment of variables 9](#_Toc95479320)

[Figure 6: Good and Bad return handling 9](#_Toc95479321)

Tables

**No table of figures entries found.**

Introduction – Tīmatanga Kōrero

Purpose

1. To provide a consistent framework from which to write, review and audit code files which assist in providing Insights to, or are the source code of products for, the National Road Policing Centre (NRPC).

Background

## Why documentation is important:

1. It is good for knowledge transfer. Not all code is equally obvious. There might be some complex algorithms or custom workarounds that are not clear enough for other developers.
2. It helps to troubleshoot production issues. If there are any problems with the product after it’s released, having proper documentation can speed up the resolution time.

Finding out product details and architecture specifics is a time-consuming task, which results in the waste of your money.

1. It may help you better manage any additional integrations and product add-ons. Product documentation describes dependencies between system modules and third-party tools. Thus, it may be needed for integration purposes.
2. For products it helps ensure that end users, and auditors, know what the system is supposed to be doing, and how that desired functionality is intended to be achieved.
   1. Further, it should provide the scope of the product’s functionality, and clearly define the environment(s) for which the product was designed, and its limitations.
   2. The documentation should also outline where human decision making influences the system’s inputs and its outputs (in line with the system’s limitations and purpose).

Guidelines for code – Aratohu mā Waehere

General Principles:

1. Keep it simple and concise. Follow the DRY (Don’t Repeat Yourself) principle. You don’t need to comment on every single line of your code, use comments to explain something that really needs explaining and is not self-evident.
2. Always keep it up to date. It’s best to document the code step by step, as it is written, instead of writing down the comments for the code that was written months ago. In doing so, you will save time and make the documentation precise and complete. Use proper versioning to keep track of all changes in the document.
3. Document any changes to your code. Documenting new features or add-ons is obvious. However, you should also document deprecated features, capturing any change in the product.
4. Use simple language and proper formatting. Code documents are typically written in English. The best practices for documentation writing require using the Imperative mood, Present tenses, preferably active voice, and second person. Use consistent header, footer, headings, and font sizes for better readability.

This ensures the transparency of the code’s processes and purpose.

1. Combine automated documentation tools and human input. Automation will speed up the process, but a person can make the code documentation comprehensible while adding more of a personal touch.

Requirements of documentation:

1. A “README” file which contains:
   1. A brief description of the project
   2. Why the project is useful
   3. Installation instructions
   4. A short example/tutorial
2. ~~Allow issue tracker for others~~
3. ~~API documentation~~
4. What a function does
5. What the function's parameters or arguments are
6. What the function returns
7. Consistent coding conventions, such as file organization, comments, naming conventions, programming practices, etc.
8. Include information for contributors
9. Include citation information
10. Include licensing information
11. Provide author contacts, i.e. e-mail address(es)
12. List all the versions of the files along with the major edits you did in each version – or use a version control system with commit messages
13. File level annotations should be placed at the top of a file
14. Package imports should be the first lines of code, followed by any global parameters.

Terminology:

|  |  |
| --- | --- |
| Line | A single line of code, with one purpose. These are typically implicitly numbered, to allow for easy debugging. |
| Execution | Running the written code, done normally across a whole file, but can be done line-by-line. |
| Debugging | The process through which code’s execution is inspected, typically for the purpose of identifying where an error or unexpected behaviour (“bug”) occurs from, and why it occurred |
| Argument(s) arg(s)) | Objects passed to a function, through a function call. Typically, these are data values, but they can be other functions (“nested function” calls). |
| Function | A set of lines of code, which together complete a more complex set of operations, for a single purpose. These typically have more complex annotations/comments, specific to their purpose and structure |
| Class | An abstract object, which contains its own data fields (values) and methods (functions), referred to collectively as “attributes” of that class |
| Object (obj) | An instance of an abstract class, i.e. “*Arlo*” is an instance of “*Police Dog*”, “*3*” is an instance of “*Integer*” |
| Variable (var) | A generic term for a “box” which can hold a value (sometimes specified to be of a specific type). i.e. aNumber = 3, this operation assigns the value “integer: *3*” to the variable “*aNumber*” |
| Parameter (param) | A value which influences the execution of a function or operation  i.e. *x* is a parameter of *y = 2\*x* |
| Comments | The method of informing authors and external viewers of a code fragment’s context or purpose. Written text within a comment are NOT executed when their parent code fragment is executed. |
| Line comment | Single line of annotations, done above a code line which the comment relates to:  *// this calculate the class average and stores it in classAverage*  *classAverage = (grade(1) + grade(2) + ... + grade(5)) / studentCount;* |
| In-line comment | Annotations provided on the same line as a code fragment:  *jims\_age = 31; // this sets jim's age to 31* |
| Block comment | A more detailed comment type, typically providing detailed context and descriptive information about a code fragment, located below the definition line of that fragment. These should be used to document functions or classes |
| Repository | Folder containing code, supporting documents, inputs and outputs |
| Codebase | All code relating to a project, found within a repository specific to that project. |
| Source code | The code which is used to create a specific product. Within a project’s “codebase”, there may be “source code” for several products. |
| Refactoring | Redesigning the code or the code base to improve readability, efficiency, or to convert code written in one language into another language for some gain/purpose. |
| Parent | A higher order thing that invokes other, lower order, things |
| Child | A thing invoked by a higher order thing |
| Main | The primary “thread” of execution. Other processes/universes branch off from this thread. |
| Global | Global attributes are accessible across the entire code file and should be located near the top of the file. |
| Local | Local attributes exist only within a class or function and are not accessible outside of that function.  These are effectively brought into existence inside a parallel universe when their parent attribute is called, and as such they cannot be accessed from any other universe, and cease to exist when that parent attribute is no longer running/deleted |

Conventions:

1. Naming
   1. Dynamic variables should start lowercase, and use camelCase
   2. Constants should user all UPPERCASE, to show they should not be changed. This is enforced with the datatype flag “FINAL” where possible
   3. Functions should use BigCamelCase, a variant of camelCase where the first character is an Uppercase character.
   4. All names should have an intelligible meaning, and where abbreviation is needed, the full-name : abbreviation pair should be written out in the file level comment
2. Comment
   1. All comments should be separated from their comment-toggle-character by a single space character
   2. In-line comments should be separated from the executable code by a single space character and should be used to explain something highly specific within the line.
      1. i.e. jimAge = 31 # this sets jim's age to 31
   3. Line comments should go above the corresponding line and provide details or context about the operations of that line.
   4. Block comments should start on a line below the toggle-character(s), and should end one line before the toggle-character(s) and should be used to explain a function or a long (multi-line) process.
      1. i.e.

"""

This is a block comment, where more detailed annotations go

You can have any number of lines of annotations

within a block comment

"""

1. Line operations
   1. Operators should be separated by a space character for visual clarity
   2. Brackets should be used for operation grouping, even when this would be implicitly accomplished. This ensures there is no accidental typecasting or operations mis-sequencing, and more clearly communicates author intent.
      1. i.e. mathOp = (x + y) \* z
   3. Listed items should be separated by a space following a comma
      1. i.e. c("red", "green", "blue")
2. Functions
   1. Function header
      1. Definition of the function call
      2. Annotations
      3. Required “arguments” of that function:
         1. :param <param name> : <what it is>
      4. Returned values from that function
         1. :return <returned obj name> : <what it is>
   2. Function body:
      1. Code lines which are run in sequence when the function is called
      2. Return, what the function spits out after its completion, there is not always a returned value
   3. i.e.

|  |  |
| --- | --- |
|  | Header: call structure  Annotations:  Purpose  Argument definition  Function body:  Executed code  nothing to return |
|  | Header: call structure  Annotations:  Purpose  Argument definition  Function body:  Executed code  Returned value |

Example documentation of specific subcategories of code:

1. Line:

|  |
| --- |
| Figure 1: Line with in-line comment |
| Figure 2: Line with above-line comment, broken up for readability |

1. Functions:

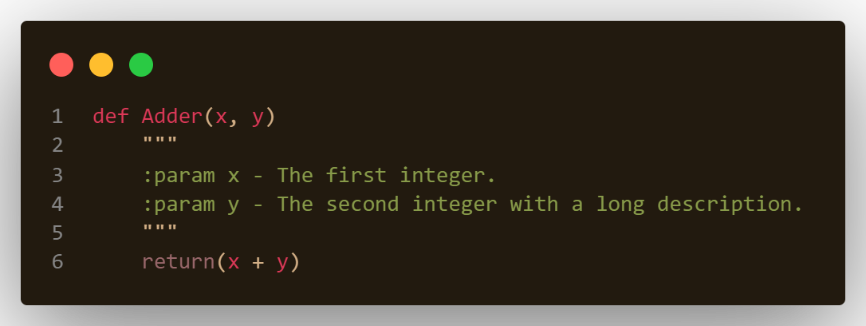


Figure 3: Basic function description

1. Class Objects:

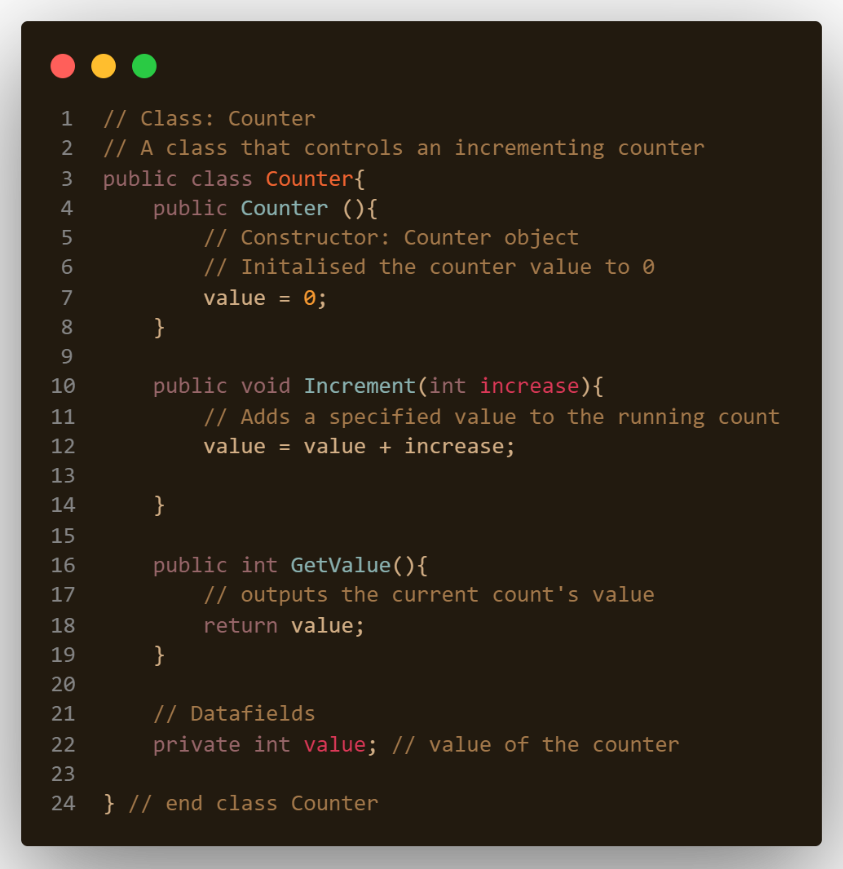


Figure 4: Class Object with annotated functions

1. Assignment:

Left-hand assignment ensures the key feature of each line is immediately visible, making it easy to see in code where an object is defined.

With long or complex calls, right hand assignment does not ensure a consistent implied grid position of assignments, making it difficult to see where objects are defined

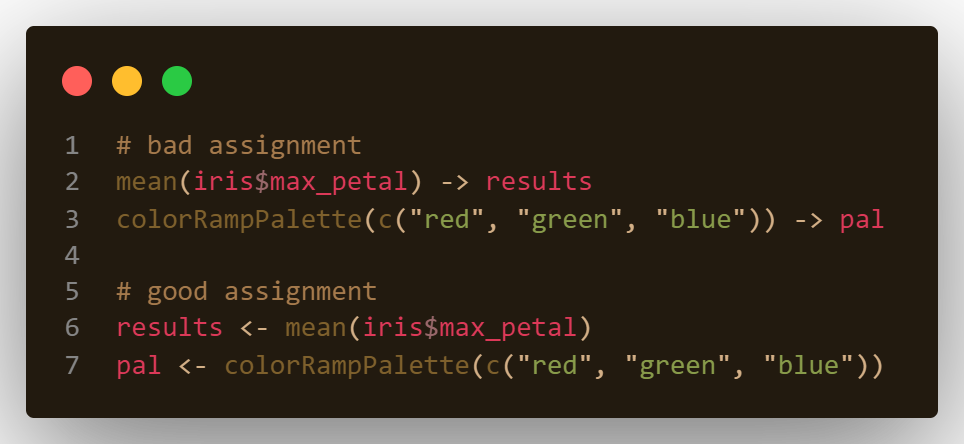


Figure 5: Good and Bad assignment of variables

1. Explicit returns:

Declaring exactly what should be returned ensures implicit return does not return something different from the expect/desired value, and ensures the codebase is consistent across different programming languages, which may handle implicit return differently.

Further, explicit return helps assist debugging, as it provides clarity of what the developer expected to be returning.



Figure 6: Good and Bad return handling

References – Tohutoro

Guides for documentation of simple code fragments

1. <https://www.naturaldocs.org/getting_started/documenting_your_code/>
2. <https://google.github.io/styleguide/Rguide.html>
3. <https://developers.google.com/machine-learning/glossary>