Programming Principles

Assignment 1 ("Individual Programming Assignment") Marking Rubric

Below is the rubric used to grade your assignment submission:

	Poor	Adequate	Good	Great
Pseudocode *Marked out of 5*	*Mark of 0 to 2* Pseudocode was not submitted, or does not adequately achieve the goals of pseudocode. The steps of the program design are not clearly and accurately described in English, and/or the pseudocode does not depict an appropriate solution to the task.	*Mark of 2.5 to 3* Pseudocode is adequate but could be improved to ensure that it clearly and accurately describes all significant steps of the program design in English. The program design is an adequate but not optimal solution to the task and/or does not match parts of the implementation.	*Mark of 3.5 to 4* Pseudocode is quite well written, describing most steps of the program design in clear and accurate English. The program design is a fairly good solution to the task and mostly matches the implementation.	*Mark of 4.5 to 5* Pseudocode is well written, clearly and accurately describing the steps of the program design in English. The program design is a good solution to the task and matches the implementation.
Functionality *Marked out of 10*	*Mark of 0 to 4.5* The source code does not implement the requirements detailed in the brief to an adequate degree. There are numerous errors or omissions, which may include the definition of functions to their specifications. The program may contain significant syntax errors or not display appropriate output.	*Mark of 5 to 6.5* The source code implements an acceptable number of the requirements detailed in the brief, with issues or omissions which may include the definition of functions to their specifications. The program may contain minor syntax errors or not display appropriate output in some circumstances.	*Mark of 7 to 8.5* The source code implements most of the requirements detailed in the brief, with some issues or omissions which may include the definition of functions to their specifications. The program contains no syntax errors and displays appropriate output in most circumstances.	*Mark of 9 to 10* The source code implements all or almost all the requirements detailed in the brief, including the definition of functions to their specifications. The program contains no syntax errors and displays appropriate output.
Code Quality *Marked out of 5*	*Mark of 0 to 2* The quality and presentation of the code does not demonstrate an adequate understanding of the concepts and commands involved, or the implementation is incomplete. There are significant issues regarding the design or efficiency of the code, and it does not adhere to best practise.	*Mark of 2.5 to 3* The quality and presentation of the source code could be improved, but overall demonstrates an adequate understanding of the concepts and commands involved. Some aspects of the implementation are inefficient or do not adhere to best practise.	*Mark of 3.5 to 4* The source code is written and presented quite well, demonstrating a good understanding of the concepts and commands involved. The implementation is quite efficient and adheres to best practise most of the time.	*Mark of 4.5 to 5* The source code is written and presented well, demonstrating a mastery of the concepts and commands involved. The implementation is efficient and adheres to best practise.

When assignment results are released, your mark in each section will be specified, alongside the corresponding feedback statement from the rubric:

```
Student
--------------------------
12345678        LOVELACE        Mrs     Ada

Pseudocode [4/5]
--------------------------
Pseudocode is quite well written, describing most steps of the program design in clear and accurate English.
The program design is a fairly good solution to the task and mostly matches the implementation.


Functionality [9.5/10]
--------------------------
The source code implements all or almost all the requirements detailed in the brief, including the definition
of functions to their specifications.  The program contains no syntax errors and displays appropriate output.


Code Quality [3.5/5]
--------------------------
The source code is written and presented quite well, demonstrating a good understanding of the concepts and
commands involved.  The implementation is quite efficient and adheres to best practise most of the time.


Total [17/20, Grade: HD]
--------------------------
Well done.
Remember to use meaningful variable names, and the function could be implemented more efficiently.
```

For demonstration purposes only, actual appearance may vary.

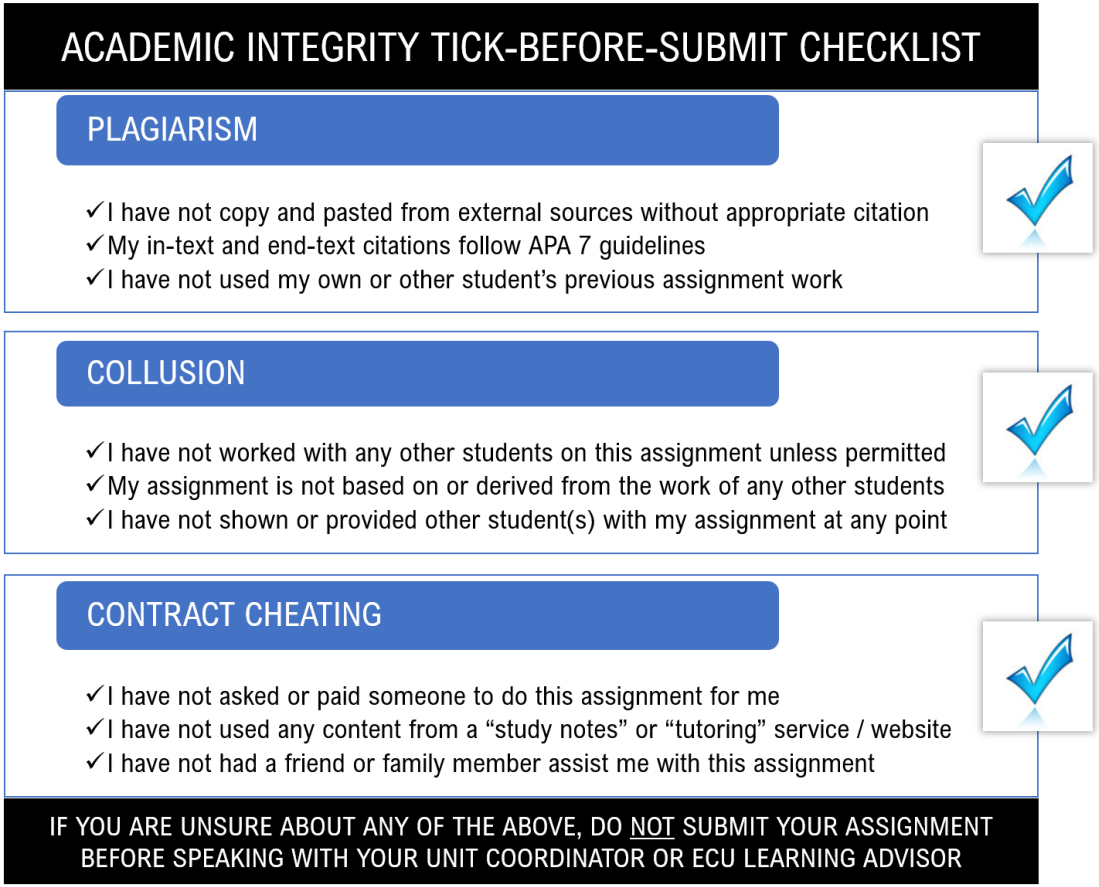
Your total mark and the corresponding grade will also be included, as well as additional comments/feedback as necessary. *If you have any questions about your mark, would like more detailed feedback, or have any concerns, please email the unit coordinator. They will be happy to provide further detail, address any concerns, and review your mark.* Before doing so, please view the list of criteria on the next page, as it may help to put your mark into context.

Tip: Check the Blackboard Discussion Board regularly for assignment-related tips, examples and advice. You can also ask questions there, but don't upload your work to the discussion board!

Email your assignment work to your tutor *at least once* while working on it, so that they can give you advice and feedback on it. This will help you to improve your work (and your mark)!

Academic Integrity

As well as reading the information regarding academic integrity in Blackboard and the assignment brief itself, be sure to watch this video and go through the checklist below *before* submitting your assignment. Academic integrity is taken very seriously, and breaches attract penalties ranging from a mark of 0 for the assignment to expulsion from the university.



List of Marking Criteria

The following is a list of criteria that staff consider when marking your assignment. It is not all-inclusive, since attempting to write such a list would not be feasible and not all criteria is applicable to all assignments, however it provides insight into the marking process. While this list can be used to put your marks into context when results are released, it is best used *prior to submission* – to assist you in completing your work to a high standard.

Pseudocode Criteria

The main resources regarding pseudocode are the introduction to it in Lecture 1, the information in Reading 3.3, and the advice in the assignment brief. The recording of the case study task of Lecture 5 contains an example of writing pseudocode for a program that is of similar scope to the assignment. Be sure to use these resources when writing your pseudocode. More detailed criteria is presented below, split into three sections.

Overarching Concepts

- Think of pseudocode as a recipe – You are writing a document that should describe how to do something in enough detail for someone else to be able to do it without further information. You are describing the program design, not what the syntax of the code looks like.
- Pseudocode should describe the steps of your program design in a way that is not specific to a particular programming language. Avoid mentioning the names of Python-specific functions (although using words like “print” and “input” in your sentences is fine – they’re generic/common words).
- Pseudocode should describe the steps of your program design in clear and complete English sentences, and not be full of the various symbols required when writing actual code. For example, “Add 1 to total” is better than “Set total = total + 1”.
- Your sentences should be precise and accurate, clearly describing what happens. Always focus upon *what* the code does – you should not need to go into detail regarding *why* it is being done since this should be self-evident in a good program design and well-written pseudocode.
- Pseudocode should be consistent wherever possible – use consistent terms and sentence structures to describe things throughout the document. For example, do not alternate between “Display…”, “Show…”, “Print…” and “Output…” when describing showing output at various points.

Presentation and Structure

- Your pseudocode document should have two main sections: The first section should be the pseudocode for the main body of the program. What it does when it is run. The second section should be the pseudocode for the functions defined in / used by the program. Think of the second section as an appendix of a book – it is there in case the reader wants further details about a topic, but not getting in the way of the main text of the book.
- The pseudocode for your functions should be clear about the parameters they define, the steps of the task they perform, and what they return.
- When a step of your main program involves calling one of your functions, mention it by name and be specific about the parameter values so that the exact relevance and usage of your functions is clear.
- Use indentation to convey the structure of your program, e.g. Indicating which steps occur in the bodies of loops and selection statements. Make sure that your indentation accurately reflects the program structure – There should not be different indentation between steps in the same block.
- There is no need to present the steps of your pseudocode as a numbered or bulleted list. Simply write the statements out in the appropriate order, using indentation to reflect the structure of the program. Bullets/Numbering takes up extra space and clutters the appearance of pseudocode.
- Pseudocode should adhere to the concepts and control structures of programming. For example, when using a loop you should introduce it with a step that describes how the iteration is controlled (e.g. “Repeat for each score in the scores list”, “While choice is not 5”, or “Loop endlessly”), followed by indented steps describing what happens inside the body of the loop.
- Avoid merging multiple distinct steps of the program into a single sentence, particularly if they involve different parts of the program structure. For example, avoid describing an “if” statement and its condition, and what happens if it is True all in a single sentence.
- Consider using a landscape page orientation to allow for longer sentences/more indentation. If a sentence gets long enough to be “wrapped” at the edge of the page, pay attention to the indentation of the wrapped portion to ensure the structure/flow of the document remains intact. Long sentences are also worth re-examining to see if they can be shortened or split into separate steps.
- There should be no need for anything like a “comment” in pseudocode. Comments are used to add extra context to actual code, which should not be needed in well-written pseudocode.

Minutiae

- Import statements (i.e. importing specific Python modules) should not be mentioned in pseudocode, since they are language-specific implementation details rather than a meaningful step in your program design.
- If your program needs variables to be initialised to a specific value at the start (typically 0, an empty string, or an empty list), be sure to mention this in the pseudocode. Knowing what these variables are initially set to is important for when they are added/concatenated/appended to later.
- When describing "if" statements, a single sentence should be used to describe the statement and the condition, e.g. "If choice is 3". Do not split it into two steps such as "Check if choice is 3" followed by "If True". The statement and the condition are one step in the program design.
- Make sure that your pseudocode distinguishes between describing separate "if" statements versus an "if" statement followed by "elif" statements. For example, use "Otherwise, if…" to describe the "elif" statements, rather than just "If…" – The resulting program structures are different!
- When describing steps that occur in the body of a loop, use singular terminology – you are describing what happens *on each iteration* of the loop.
- In some situations it is appropriate to simply describe output, e.g. "Print a confirmation message", but at other times it may be relevant to be more specific – providing details or the exact text. Think about what would be important to someone implementing the program using your pseudocode.

Functionality Criteria

The program requirements and details of functions/methods in the assignment brief dictate the requirements of the assignment. Be sure to read and re-read them, and double-check them before submitting your work. Even if you feel that you have understood and implemented everything well, it is worth sending your work to your tutor for some feedback prior to submitting it. Writing a list of functionality criteria is difficult, since so much of it is specific to the assignment, but the following list describes some generic criteria to check against your work.

- Code containing syntax errors cannot be run, and will lose marks in the functionality section. If the errors are minor, staff may fix/bypass them to continue marking, however if they are significant then your work will be marked by looking over the code and awarding partial marks where appropriate.
- Code that is written in Python 2 will not run in Python 3, and will lose marks in the functionality section. There is no reason or excuse to use Python 2.
- Be sure to read (and implement) all parts of each requirement in the assignment brief – both the numbered points and the dot points beneath them. Some aspects of the functionality are detailed in the dot points and can be missed if not reading closely.
- There is always more than one way to implement a program, but please don't stray too far from the requirements when implementing your assignment. The approach it details aims to strike a balance between being a good approach, being appropriate for the unit, and testing the necessary unit content.
- If you want to add something extra to your work, always be sure to implement the requirements before upon adding extra things, and make sure that you are *expanding upon* the core requirements, rather *changing* them. Always consult with your tutor beforehand – we love creativity, but must mark to the rubric.
- The assignment will require you to define at least one function/method (methods are relevant to Assignment 2 only). The brief will be very clear about the parameter(s) that the function/method receives (if any), what the function does, and what it returns (if anything). Adhering to these specifications is important. Marks will be lost if the function is not defined, does not receive/use parameters as specified, does not do what is specified, or does not return the specified value.
- Printing output to the user is an important part of every assignment. While your program's output does not need to exactly match screenshots in the brief (unless specified in the brief), make sure that it conveys the same information and is presented well.

Code Quality Criteria

While the functionality of your program is mainly assessed by running it, the quality of your code is assessed by looking at the code. There is quite a lot of criteria considered here, drawn from all aspects of the unit content. Note that when it comes to concepts such as "best practise" and "efficiency", we are considering such things based upon what has been covered in the unit – i.e. Our expectations are based upon what a student could reasonably be expected to implement based upon unit content up to that point. Some of the criteria involved in evaluating the quality of code includes…

Understanding and Application of Concepts

- Python is a dynamically typed language – you do not need to create or initialise a variable before assigning a value to it. The only time you must create a variable in advance is when it is referred to before a value is assigned to it, e.g. setting "total" to 0 so that it exists before trying to add something to it.
- You should always be aware of the data type of each value/variable in your program, including the data type of the return value of a function. Hence, you should only use functions like "int()", "str()" or "float()" when they are actually needed, to convert a value from one data type to another.
- Remember, the "input()" function always returns a string – wrapping "str()" around it is not necessary.
- Always try to store a variable in the data type that best represents the value / allows it to be used the most conveniently later in your program.
- Use a "while" loop when you need to repeat something an unknown number of times, e.g. if it depends upon user input or a random number, and use a "for" loop when you need to repeat something a specific number of times or work your way through the values in a data structure.
- The "enumerate()" function can be used in a "for" loop to give you a variable containing the index number of the current item as well as a variable containing the value of the current item as you loop through a data structure (see Lecture 3). This often eliminates the need for a "counter variable" to be created/incremented.
- The main place you are likely to use the "range()" function is in a "for" loop, to make it repeat a certain number of times. If you find yourself using "range()" in an "if" or "elif" condition, what you are doing can almost certainly be done more efficiently and reliably using a comparison such as <, >, <= or >=.
- Use an "else" rather than a second "if" statement or an "elif" in situations where it is appropriate to do so. You should never need to have a second comparison checking for the opposite of the previous comparison.
- Do not use the "continue" statement unless you need to – If the next thing that would happen in your code is that it would reach the end of the loop body, you don't need a "continue" statement at that point. Always keep track of the structure of your code amongst all the loops and selection statements!
- The "del" command is the most appropriate way to delete a specific item from a list by its index number. The ".pop()" list method is only useful if also want a copy of that item returned at the same time.

Code Layout and Readability

- Use a blank line or two to separate sections of your code. This increases the readability of the code, making it easier to identify different sections.
- Be sure to include appropriate commenting. This mainly involves "metadata" comments at the start of the program (e.g. your name and student number), comments summarising functions you've defined, comments indicating the main sections of the code, and comments explaining particularly complex lines of code.
- That said, if you have particularly complex lines of code, always consider whether they could be rewritten to be clearer.
- Having *too much* commenting can hinder the readability of code. Always assume anyone reading code knows how to program, so you do not need to explain things that are easy to understand by reading the code.
- Try to avoid having extremely long lines of code. Such lines can typically be split up across multiple lines or statements, or broken into multiple steps.
- Take care to proofread your strings, comments and variable names so that your program does not contain spelling mistakes. Also take care with things like spacing, line breaks and capitalisation. There's a reading in Module 1 about the "print()" function that may help you to better control spacing when printing.
- Avoid printing an entire list or dictionary as output in your program, since it will include the brackets and commas around the items resulting in unattractive output. Refer to specific indexes/keys that you wish to print, loop through a data structure to print each item in it.
- Remember that you can include "\n" in a string for a line break (new line), e.g. e.g. "print('\nHello\n')". This is preferable to using empty "print()" statements.
- Use meaningful variable names - Overly generic short names make it difficult to understand the variable's purpose and makes code harder to understand.

Functions and Complexity

- Functions should represent well-defined tasks, have intuitive input and output, and ideally be relevant to multiple points of your program. A function that simply *moves* a chunk of code that is very specific to one point of one program is likely to be adding complexity without gaining the benefits that functions offer.
- A function should increase the efficiency, robustness, code reuse, readability or modularity of a program. If it doesn't do so, it may not be aiding the program. When starting out, it can be best to solve the problem first, and *then* look for areas that can be improved by introducing additional functions.
- Your functions should be defined at the start of the program and called where needed in your code. Do not define functions in the middle of your code.
- A function that calls itself is known as a "recursive" function, and is almost universally a less elegant and efficient approach than what you can do with a loop.
- There is no need to put the body of your program into a "main()" function - see the reading in Module 4 regarding this.
- Your functions should never use or rely upon global variables. Any data they require from the main program should be passed to the function via parameters, and results produced by the function should be sent back to the main program via a return statement.
- Remember to Keep It Simple. Avoid creating a solution that is over-engineered or unnecessarily complex given the scope of the problem. If you've implemented a more complex solution than necessary, you should always be able to identify tangible benefits to justify the approach, e.g. efficiency, robustness, scalability…