COMP 1510 Assignment #4:
Make me a game, please!

Christopher Thompson
chris_thompson@bcit.ca

Due **Sunday April 9th at or before 23:59:59 PM**

Games

In the “olde dayes” of internet and computing, we didn’t have amazing, graphically intense MMOs like WoW. We used to dial into things called multi-user dungeons (MUDs).

MUDs were text-based, real-time, role-playing adventure games that let users read descriptions about rooms, objects, other players, non-player characters, etc., and perform actions and interact with the virtual world. Oh my gosh, when I think of the hours and hours and hours I spent wandering those dungeons...

For your fourth assignment, you will implement a simple MUD. Of course, we don’t know how to use Python for networking yet, so this will in fact be a single-user dungeon, or a “SUD".

Command line programs are fun, but they only go so far. You are ready for the next step. For A4, you must implement a simple desktop client aka a simple graphical user interface (GUI) for your game. You must use Tkinter, and lab during week 13 will intoduce all the Tkinter you need to know.

Any project of this magnitude requires some planning. Your fourth assignment this term begins with some planning. Let’s get started.

1 Submission requirements

1. This assignment is due no later than **Sunday April 9th at or before 23:59:59 PM**.
2. **You must invite me to collaborate no later than 23:59:59 on Friday March 24th 2023**. On GitHub, I am known as chris-thompson. You will recognize my avatar.
3. **Late submissions will not be accepted for any reason. After Sunday April 9th you must shift gears and study for finals**.
4. **This is an individual or a partner assignment**. If you want to work with a partner you must both consent in writing in Slack before the end of Friday March 24th 2023 and invite me to collaborate to a repo that already has two developers.
5. I strongly encourage you to share ideas and concepts, but sharing code or submitting another person code is not allowed.

2 Grading scheme

1. A detailed grading rubric for this project has been appended to the end of this document. It includes a list of required elements you must include in your project.

3 Assignment 4 prep

1. **For assignment 4, I would like you to create your own PyCharm project**. It is time, padawan. Add git to it from the PyCharm main menu (ask me to remind you how to do this in lab!), push it to the cloud, and invite me (and your partner) to collaborate.

2. **You must invite me to collaborate no later than 23:59:59 on Friday March 24th 2023**. On GitHub, I am known as chris-thompson. You will recognize my avatar.

3. Make sure you include a README.md and the .gitignore file which you can copy from previous labs.

4. Commit and push. Hello Chris!

5. **Stop what you're doing. Before you type a single character of code, you must try playing a MUD first**. There are all sorts of resources and lists online for finding MUDS. I've chosen an old and venerated MUD for you to try called **Aardwolf**. An aardwolf is a kind of hyena that eats insects. I have no idea why the developers chose this name.

6. **Start by downloading Mudlet** from `https://www.mudlet.org/`. It is a free and helpful MUD client I've chosen for you. It is available for Windows and macOS. Download and install it please. Pay very close attention to the interface it offers you.

7. Start Mudlet. When Mudlet opens, a modal window invites you to select an existing game or create a new profile. **Create and save a new Profile for the Aardwolf MUD**:

   (a) Profile name: Aardwolf, of course!

   (b) Server address: aardwolf.org

   (c) Port: 23

8. Then log in and play!

9. Take your time, read everything, and follow the instructions. You will start by typing NEW for new character, and then you will be asked for your character name and a password. Don't lose these!

10. **Play Aardwolf until you reach level 10**. You must add a screenshot to your A4 project (in PDF format or some other easily-viewed format) to prove your character has reached 'Level 10'. You will understand what this mean after you have played for a few minutes. And how cool is this – you have to play a game for marks! **Add the PDF to your PyCharm project and commit and push using the commit message: "ACHIEVED LEVEL 10, CHRIS!"**.

11. Be kind and thoughtful when you are logged in. Several terms ago, the Aardwolf mods banned DTC BCIT IP addresses because some students were being inappropriate while playing from campus. Having the ban lifted was not easy. Please do not do this. These people take their role-playing very (perhaps too!) seriously.

12. Based on your experience with Aardwolf, apply what you have learned so far in COMP 1510 and design a very simple game. Examine the list of game requirements and the list of Python language requirements in the next section. **Develop a simplified PDF flowchart called game.pdf that captures the key elements of the game**. Show me what happens when I play your game. **Add the PDF to your PyCharm project, and commit and push using the commit message: "Complete game concept, Chris!"**.

4 Include these mandatory elements

Implement an interesting Single User Dungeon (SUD). Using your experience with Aardwolf as an example and your personal gaming, literature, or popular culture interests as a starting point, construct a simple text-based adventure experience for me. Do please keep this simple though. Remember the Zen of Python. Simple is better than complex:

1. **There must be a main function in a file called game.py**, but you are encouraged to use multiple source files. If I execute game.py as a module, the main function must permit me to play your game on the command line.

2. Complete a flowchart called **game.pdf** that is updated to correctly show me EXACTLY what is happening in the final version of your game.

3. **Your game must include the following elements**:

   (a) a whimsical, descriptive, and engaging scenario

   (b) a **5 x 5 grid-based environment** if you are working alone, or a **10 x 10 grid-based environment** if you are working with a partner

   (c) a character who has a name, health/happiness/hit points and/or other meaningful measurable attributes, a level, and abilities that improve when the character levels up

   (d) character movement in the four cardinal directions: north/south/east/west

   (e) each time the character moves, there is a chance that they will encounter some sort of obstacle when they arrive at the next set of coordinates

   (f) when the character encounter an obstacle, they must be given an opportunity to overcome it (battle/riddle/potion/tool etc. be creative!)

   (g) the game ends when the character achieve a final goal

4. **Develop a simple leveling scheme**:

   (a) My character should start at level 1, and be able to reach level three (3).

   (b) Each level needs a name, and a certain amount of experience or puzzle solving need to be accrued before the character reaches that level.

   (c) When a character reaches a new level, two things should happen – the maximum health/happiness/hit points and/or other meaningful measurable attributesshould increase some reasonable amount, and something about the character must improve.

   (d) When a character reaches level 3, they should finally be able to take on the boss aka the final task with a reasonable expectation for success.

5. **Create a coherent, rich ecosystem of challenges**. Perhaps foes become more challenging as the character level increases. Maybe your game space will have regions for different levels. Be creative and consistent.

6. If a character runs out of mojo, the game must end.

7. **Your game must incorporate the following Python elements**:

   (a) use of immutable data structures like tuples to minimize unnecessary mutability

   (b) use of mutable data structures like lists and dictionaries in a thoughtful and correct manner, i.e., no unnecessary looping through dictionaries

   (c) thoughtful use of exceptions and exception handling that prevents the program from crashing

   (d) minimized scope and lifetime of all variables and objects

   (e) decomposition of your idea into a collection of small, atomic, independent, and reusable functions

   (f) simple flat code that is easy to understand

   (g) demonstration of an understanding of how comprehensions work through the meaningful and correct use of one or more list/dictionary comprehensions

   (h) selection using if-statements

   (i) repetition using the for-loop and/or the while loop where it makes sense but not excessively

   (j) use of the membership operator where it makes sense

   (k) appropriate use of the the range function

(l) thoughtful and meaningful use of one or more functions from itertools

(m) the random module

(n) function annotations

(o) doctests and/or unit tests for every single function (that is, every function needs doctests or unit tests or doctests and unit tests).

(p) ALL output must be formatted using f-strings and/or str.format and/or %-formatting

8. Once your command line game works, you must augment your game. **Use Tkinter to implement a graphical user interface (GUI) for your game**, a "SUDlet," if you will. This must be implemented in a file called game_gui.py. This file must contain a main method that lets me play your game using the GUI.

9. That's it. Keep it simple. I want clean code, short functions, lots of comments, and simple game play.

10. I will withhold marks for solutions that are unnecessarily complicated. Simplicity and clarity trump everything. Top marks will be reserved for games that are short and sweet and elegant.

11. No global variables. Global constants are permitted. Absolutely no single letter variable names.

5 Style Requirements

1. You must observe the code style warnings produced by PyCharm. **You must style your code so that it does not generate any warnings.**

   When code is underlined in red, PyCharm is telling us there is an error. Mouse over the error to (hopefully!) learn more.

   When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more.

2. **You must comment each function you implement with correctly formatted docstrings.** Include informative doctests **where necessary and possible** to demonstrate to the user how the function is meant to be used, and what the expected reaction will be when input is provided.

3. In Python, functions must be atomic. They must only do one thing. If a function does more than one logical thing, break it down into two or more functions that work together. Remember that helper functions may help more than one function, so we prefer to use generic names as much as possible. Don't hard code values, either. Make your functions as flexible as possible.

4. Don't create functions that just wrap around and rename an existing function, i.e., don't create a divide function that just divides two numbers because we already have an operator that does that called / .

5. **Ensure that the docstring for each function you write has the following components (in this order)**:

   (a) Short one-sentence description that begins with a verb in imperative tense and ends with a period.

   (b) One blank line

   (c) Additional comments if the single line description is not sufficient (this is a good place to describe in detail how the function is meant to be used)

   (d) One blank line if additional comments were added

   (e) PARAM statement for each parameter which describes what the user should pass to the function

   (f) PRECONDITION statement for each precondition which the user promises to meet before using the function

(g) POSTCONDITION statement for each postcondition which the function promises to meet if the precondition is met

(h) RETURN statement which describes what will be returned from the function if the preconditions are met

(i) RAISES statement for each Exception which the function will explicitly raise (the example here is not in fact explicitly raised, there is no raise statement, but I want you to see how it looks anyway!):

```
def my_factorial(number):
    """
    Calculate factorial.

    A simple function that demonstrates good comment construction.

    :param number: a positive integer
    :precondition: number must be a positive integer equal to or
                   greater than zero
    :postcondition: calculates the correct factorial
    :return: factorial of number
    :raises ValueError: if number is not integral or is negative
    """
    if number == 0:
        return 1
    else:
        return number * factorial(number - 1)
```

Hints and ideas

1. The main function should invoke a game function. And the game function should run the game. The “primary game loop” goes inside the game function.

2. The game() function probably accepts no parameters. Calling the game function is most probably the only line of code in your main function.

3. Inside the game() function, you need a game loop. Take a peek at this snippet of useful code. It’s a simple game loop you can use to inspire your game:

```
def game(): # called from main
    rows = 5
    columns = 5
    board = make_board(rows, columns)
    character = make_character("Player name")
    achieved_goal = False
    while not achieved_goal:
        // Tell the user where they are
        describe_current_location(board, character)
        direction = get_user_choice( )
        valid_move = validate_move(board, character, direction)
        if valid_move:
            move_character(character)
            describe_current_location(board, character)
            there_is_a_challenge = check_for_challenges()
            if there_is_a_challenge:
                execute_challenge_protocol(character)
                 if character_has_leveled():
                     execute_glow_up_protocol()
            achieved_goal = check_if_goal_attained(board, character)
        else:
            // Tell the user they can’t go in that direction
    // Print end of game stuff like congratulations or sorry you died
```

Look at this code snippet carefully, Note what I have done here. I have assembled the parts of the game, i.e., a board that is 5 by 5, and a character. I’ve also declared a variable called achieved_goal and set it to False. We will not change this variable to True until the character finishes the game.

4. Implement a function called make_character. This function can create and return a dictionary that contains key:value pairs, perhaps something like this:

```
>>> player = make_character("Chris"’)
>>> player
{"Name" : "Chris", "X-coordinate": 0, "Y-coordinate": 0, "HP": 5, "Max HP": 5}
```

5. Implement a function called make_board. This function can accept two positive, non-zero integers called rows and columns. This function is probably not required to behave correctly if it receives anything else. This function can create and return a dictionary that contains rows * columns keys, where each key is a tuple that contains a set of coordinates, and each value is a short string description. I made four empty rooms here, but maybe you can use a list of interesting descriptions and randomly assign one to each location. For example:

```
>>> rows = 2
>>> columns = 2
>>> board = make_board(rows, columns)
>>> board
{(0, 0): ‘Empty room’, (0, 1): ‘Empty room’, (1, 0): ‘Empty room’, (1, 1): ‘Empty room’}
```

6. Inside the game-loop, the first thing we do is describe the current location. Implement a function called describe_current_location. My version of this function accepts the board and the character, but you may decide to make your function a bit differently. You can use the character’s coordinates to retrieve the information about that location from the board dictionary, and print the description.

7. The next thing we do in the loop is ask the user where they wish to go. Create a function called get_user_choice. This function can print an enumerated list of directions and ask the user to enter the letter or number corresponding to the direction they wish to travel, and return the direction. Choose a system and stick to it, i.e., North-East-South-West, or Up-Down-Left-Right. Don’t let me enter junk. If I enter something that is not a number from the list, make me choose again. Reject all user input that is not correct. Tell me to try again. And again. And again. Keep looping while my input is not correct. Also I hate typing, so make my choices single letter or number choices.

8. The user is not allowed to cross the boundaries of the game board. Implement a function called validate_move. The version I’ve created here accepts three parameters, but yours may be a little different. This function can determine where the player is on the board and whether they can, in fact, travel in their desired direction. If so, return True. If not, return False. You will need to use some clever arithmetic here to ensure I don’t go off the board!

9. If the move is valid, you can invoke a function called move_character which accepts the character dictionary and updates the character’s X- and Y-coordinates appropriately.

10. After the character has moved, describe the new location. Then check to see if they have reached a challenge. Maybe this will happen every single move, or maybe it will happen somewhat randomly. You decide!

11. What happens when I encounter a challenge? I have to overcome it. Will it be a riddle? A puzzle? Combat? You decide!

12. I left out one crucial thing. What if I die? What if I run out of health or happiness or hit points etc.? Modify the guard condition in the while loop. Ensure game play ends when my HP reaches zero, even if I have not reached the end of the game. Perhaps you will make a function called is_alive which accepts the character and returns True if my HP is not 0, else False, like this:

```
while is_alive(character) and not achieved_goal:
```

13. There is absolutely nowhere in this assignment where recursion makes sense. Functions should not endlessly call other functions. Things need to return to the main loop in the game function.

14. The character needs to store a lot of information. I would use a dictionary.

15. Your challenges should be interesting. There are no limits here. The only requirement is that I must encounter a final grand boss or challenge at the end of the game. It must be reasonable for me to defeat the boss, I do not want to be marking your game for hours and hours! When I defeat the boss the game must end. Gracefully. With fanfare. Maybe some cool ASCII art.

16. Devising unique situations for each of 5 x 5 = 25 or 10 x 10 = 100 game cells (locations) in a huge list is not something I want you to do. Be creative. Create a data structure that maps coordinates to functions. The functions can autogenerate some sort of location, maybe using randomly selected bits of different lists of environment descriptions... Just thinking out loud. I did not say that a person needs to be able to trace their steps. That's it. Good luck, and have fun!

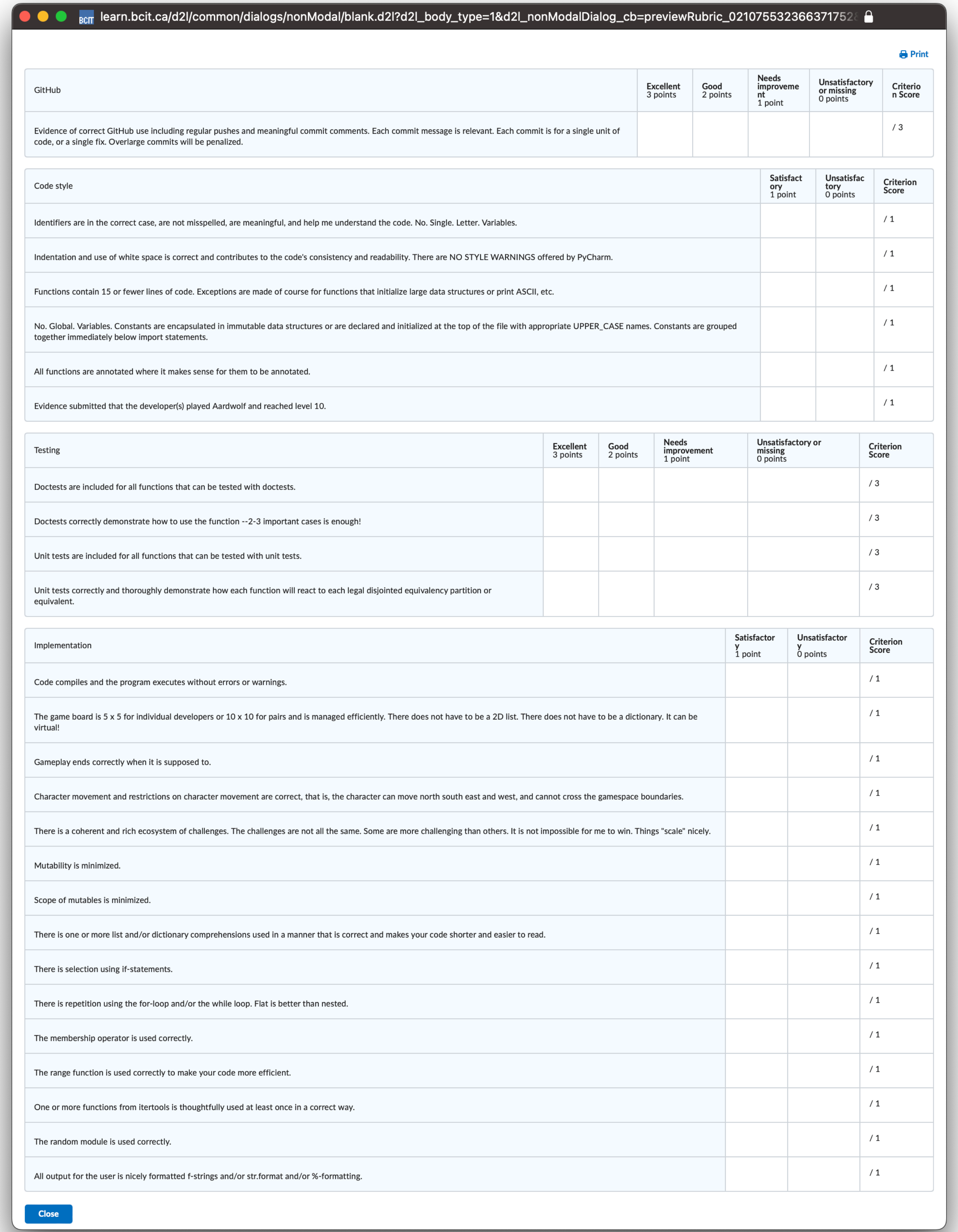
17. The highest marks will be reserved for exemplary submissions that demonstrate elegance and parsimony of code, i.e., short functions aka the fewest number of functions possible aka the fewest number of lines of code in game.py.

18. Do not make me type things. I want to select from numbered lists as much as possible.

19. The GUI can be as simple as a widget that shows me the game output and a textbox that lets me enter my choices. A map of the surroundings would be cool. Seeing my character's stats being updated as I play would also be cool. Play with Tkinter and show me what you can do!

If you're not interested in Dungeons and Dragons, make me a Hello Kitty Online Adventure. Or a Star Trek Adventure. Or a RuPaul's Drag Race Adventure. Be creative. The only limit is your imagination.

Good luck, and have fun!


learn.bcit.ca/d2l/common/dialogs/nonModal/blank.d2l?d2l_body_type=1&d2l_nonModalDialog_cb=previewRubric_021075532366371752

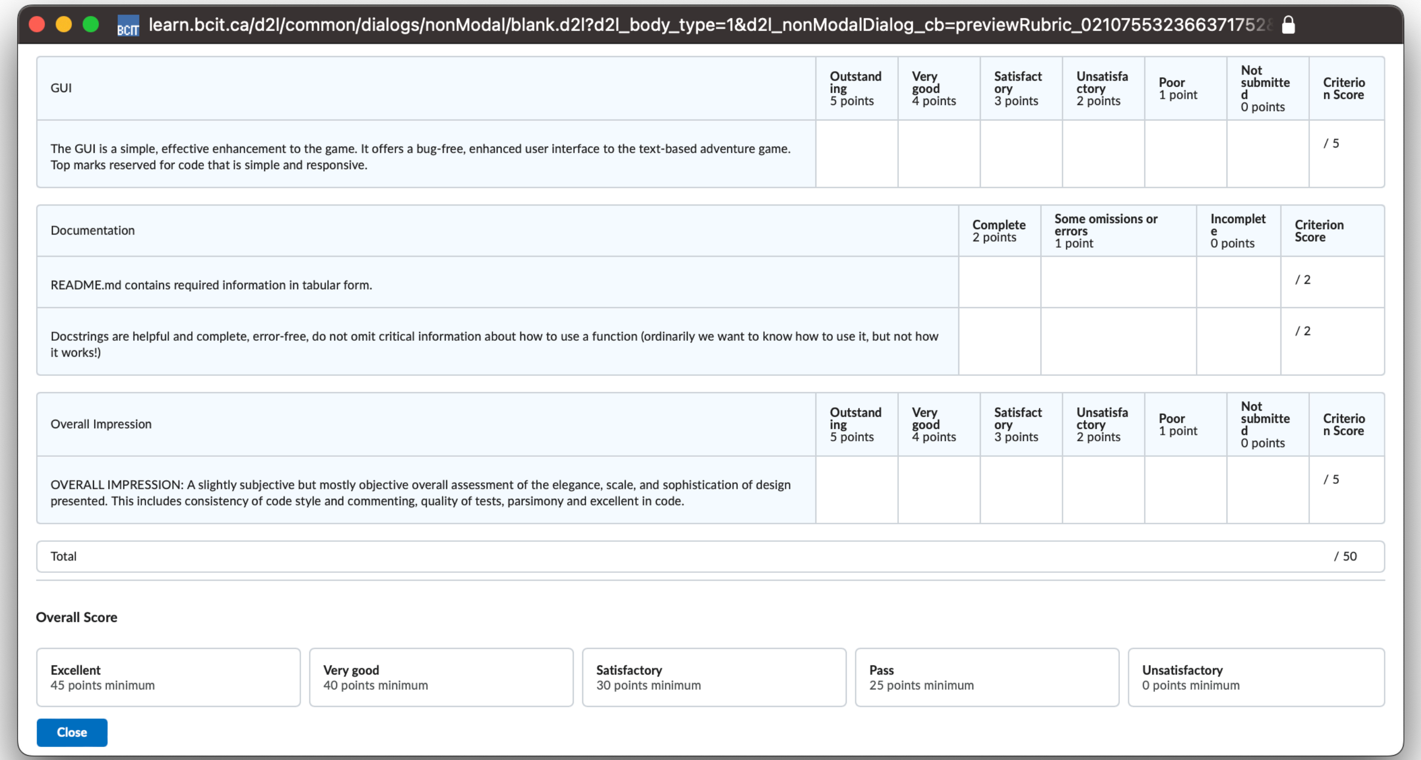
Print

GitHub	Excellent 3 points	Good 2 points	Needs improvement 1 point	Unsatisfactory or missing 0 points	Criterion Score
Evidence of correct GitHub use including regular pushes and meaningful commit comments. Each commit message is relevant. Each commit is for a single unit of code, or a single fix. Overlarge commits will be penalized.					/ 3

Code style	Satisfactory 1 point	Unsatisfactory 0 points	Criterion Score
Identifiers are in the correct case, are not misspelled, are meaningful, and help me understand the code. No. Single. Letter. Variables.			/ 1
Indentation and use of white space is correct and contributes to the code's consistency and readability. There are NO STYLE WARNINGS offered by PyCharm.			/ 1
Functions contain 15 or fewer lines of code. Exceptions are made of course for functions that initialize large data structures or print ASCII, etc.			/ 1
No. Global. Variables. Constants are encapsulated in immutable data structures or are declared and initialized at the top of the file with appropriate UPPER_CASE names. Constants are grouped together immediately below import statements.			/ 1
All functions are annotated where it makes sense for them to be annotated.			/ 1
Evidence submitted that the developer(s) played Aardwolf and reached level 10.			/ 1

Testing	Excellent 3 points	Good 2 points	Needs improvement 1 point	Unsatisfactory or missing 0 points	Criterion Score
Doctests are included for all functions that can be tested with doctests.					/ 3
Doctests correctly demonstrate how to use the function --2-3 important cases is enough!					/ 3
Unit tests are included for all functions that can be tested with unit tests.					/ 3
Unit tests correctly and thoroughly demonstrate how each function will react to each legal disjointed equivalency partition or equivalent.					/ 3

Implementation	Satisfactory 1 point	Unsatisfactory 0 points	Criterion Score
Code compiles and the program executes without errors or warnings.			/ 1
The game board is 5 x 5 for individual developers or 10 x 10 for pairs and is managed efficiently. There does not have to be a 2D list. There does not have to be a dictionary. It can be virtual!			/ 1
Gameplay ends correctly when it is supposed to.			/ 1
Character movement and restrictions on character movement are correct, that is, the character can move north south east and west, and cannot cross the gamespace boundaries.			/ 1
There is a coherent and rich ecosystem of challenges. The challenges are not all the same. Some are more challenging than others. It is not impossible for me to win. Things "scale" nicely.			/ 1
Mutability is minimized.			/ 1
Scope of mutables is minimized.			/ 1
There is one or more list and/or dictionary comprehensions used in a manner that is correct and makes your code shorter and easier to read.			/ 1
There is selection using if-statements.			/ 1
There is repetition using the for-loop and/or the while loop. Flat is better than nested.			/ 1
The membership operator is used correctly.			/ 1
The range function is used correctly to make your code more efficient.			/ 1
One or more functions from itertools is thoughtfully used at least once in a correct way.			/ 1
The random module is used correctly.			/ 1
All output for the user is nicely formatted f-strings and/or str.format and/or %-formatting.			/ 1



learn.bcit.ca/d2l/common/dialogs/nonModal/blank.d2l?d2l_body_type=1&d2l_nonModalDialog_cb=previewRubric_02107553236637175

GUI	Outstanding 5 points	Very good 4 points	Satisfactory 3 points	Unsatisfactory 2 points	Poor 1 point	Not submitted 0 points	Criterion Score
The GUI is a simple, effective enhancement to the game. It offers a bug-free, enhanced user interface to the text-based adventure game. Top marks reserved for code that is simple and responsive.							/ 5

Documentation	Complete 2 points	Some omissions or errors 1 point	Incomplete 0 points	Criterion Score
README.md contains required information in tabular form.				/ 2
Docstrings are helpful and complete, error-free, do not omit critical information about how to use a function (ordinarily we want to know how to use it, but not how it works!)				/ 2

Overall Impression	Outstanding 5 points	Very good 4 points	Satisfactory 3 points	Unsatisfactory 2 points	Poor 1 point	Not submitted 0 points	Criterion Score
OVERALL IMPRESSION: A slightly subjective but mostly objective overall assessment of the elegance, scale, and sophistication of design presented. This includes consistency of code style and commenting, quality of tests, parsimony and excellent in code.							/ 5

Total	/ 50

Overall Score

Excellent	Very good	Satisfactory	Pass	Unsatisfactory
45 points minimum	40 points minimum	30 points minimum	25 points minimum	0 points minimum

Close