

Assignment 1

Due Jan 31 by 1pm **Points** 5

Recent updates

17 Jan [MarkUs is available!](https://markus108.teach.cs.toronto.edu/csc108-2020-01/en/main) [_\(https://markus108.teach.cs.toronto.edu/csc108-2020-01/en/main\)](https://markus108.teach.cs.toronto.edu/csc108-2020-01/en/main)


16 Jan ~~The assignment submission system, MarkUs, will be available soon. We will send an announcement when it is ready.~~

CSC108H Assignment 1

Deadline: Fri 31 Jan by 1:00pm

Late policy: There are penalties for submitting the assignment after the due date. These penalties depend on how many hours late your submission is. Please see the syllabus on Quercus for more information.

Goals of this Assignment

- Use the [Function Design Recipe](#)  to plan, implement, and test functions.
- Write function bodies using variables, different types (numeric, boolean, string), and conditional statements. You can do this whole assignment with only the concepts from Weeks 1, 2, and 3 of the course.
- Learn to use Python 3, Wing 101, provided starter code, a checker module, and other tools.

Wordlock: A Word Game

In this assignment, you will write functions to implement a puzzle game that requires the player to use simple moves to reconstruct a scrambled word.

Start by watching this video about the game, in which a player gets a scrambled word and they have to rearrange the letters in the word following a set of rules. Once you've watched a couple turns, pause the video and come back here to read about the game mechanics. Then back to the video to finish.

<https://www.youtube.com/watch?v=DVTjrjRJ4Fk> [_\(https://www.youtube.com/watch?v=DVTjrjRJ4Fk\)](https://www.youtube.com/watch?v=DVTjrjRJ4Fk)



(<https://www.youtube.com/watch?v=DVTjrjRJ4Fk>)

Game Mechanics

These game mechanics describe the operations that you will solve using functions, many of which contain if statements. Read these mechanics carefully and review this section when you try to figure out example function calls — what information does each function get passed, what information does it return, and how do you gather that information?

The original string has been scrambled by first splitting it into a number of sections, each with the same length, and then rearranging each section in a random fashion, as in Figure 1. The player's objective is to unscramble the word using as few moves as they can.



Figure 1: Strings are scrambled in sections.

The player's objective is to unscramble the word using as few moves as they can. The player has three moves available: *Rotate*, *Swap*, and *Check*. The moves can only be applied to the string one section at a time.

The sections in the string are numbered 1 to N, where N is the number of sections. The section length is fixed for the game, and each section will have the same length.

For example, given the string 'wordlockgame' and a section length of 4, section 2 would be 'lock'.

You can assume that the answer length is a multiple of the section length.

The *Rotate* move is a circular shift of the string to the right, while the *Swap* move exchanges the positions of the first and last characters of the string. Both these moves are illustrated in Figure 2 below:

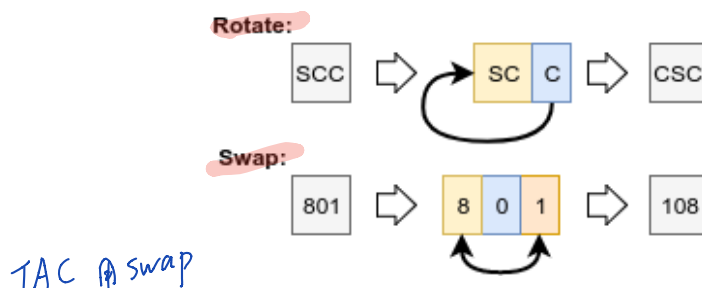


Figure 2: Rotate and Swap moves

The *Check* move is the only move that does not modify the game's state. It is used to check if a given section is correctly arranged.

The game allows for 3 modes: *Easy*, *Hard*, and *Test*. *Hard* mode involves playing the game as we have described it above, while *Easy* mode allows the player to receive hints on what sections and moves to choose each round. Enabling *Test* mode causes the program to provide the final answer before the game begins.

Gameplay

When the game begins, the player is first asked to choose a game mode. Then, the player is repeatedly asked to provide:

1. The section number corresponding to the section they want to manipulate. A string with N sections will have its sections numbered 1 to N.
2. The move they would like to apply to the section.

If the player is playing in *Easy* mode, they will be offered a hint for a section to choose, or for a move on a particular section. Getting a hint also counts towards the player's total number of moves.

The game continues until the player has unscrambled the string. We will use the phrase *game state* to refer to the current state of the string the player is trying to unscramble. When the game is over, the player's total number of moves is reported.

Starter code

写6个function

You will write code in a file called `wordlock_functions.py`. It is part of a set of files. Please download the [Assignment 1 Files](#) and extract the zip archive. Leave the files where they are, they need to be in the same directory to work properly on your computer.

文件夹推上桌面
wordlock function
wordlock game
function里写
游戏wing

Here are short descriptions of the files:

- `wordlock_functions.py`

This the file in which you will do all your work!

This file contains some [constants](#) and a complete function header and docstring (but not body!) for the first function you are to write. You will edit this file to design and write a half dozen functions, described below. For each function, you will include a function header (the `def` line, including the type contract), a function docstring containing a description and two examples, and the body of the function.

Tip: When you have written all of the functions, you might try playing the game with more challenging puzzles by changing the constants `ANSWER` and `SECTION_LENGTH` and making sure your game still works.

- `wordlock_game.py`

This is the main program. Open this in Wing and run it to start the game.

This program calls your functions that you wrote in `wordlock_functions.py`. Do not make any changes to the `wordlock_game.py` file.

- `a1_checker.py` 格式错误

We provide a checker program that you should use to check your code. See [below](#) for more information about `a1_checker.py`.

- `checker_generic.py`

This is the part of the checker that is reusable across all the assignments. `a1_checker.py` uses this to do some of its work, much like the main program, `wordlock_game.py`, uses `wordlock_functions` to do some of its work.

- `pyta`

This is a directory of a large program that examines your code for style and other issues. `a1_checker.py` uses this to do the bulk of its work.

Constants

Constants are special variables whose values do not change once assigned. A different naming convention (uppercase pothole) is used for constants, so that programmers know to not change their values. For example, in the starter code, the constant `SECTION_LENGTH` is assigned the value 3 at the beginning of the module. That value should never change in your code. When writing your code, if you need to use the value of the section length, you should use `SECTION_LENGTH`. The same goes for the other constant values.

Using constants simplifies code modifications and improves readability. If we later decide to use a different tweet length, we would only have to change the length in one place (the `SECTION_LENGTH` assignment statement), rather than throughout the program.

What to do

section - length = 3

不用 section - length

In file `wordlock_functions.py`, complete the following function definitions. Use the [Function Design Recipe](#), including writing complete docstrings for each function. We have included the type contracts in the following table; please read through the table to understand how the functions will be used.

You will note that we give you the function names and type contracts.

Remember to come up with your example function calls first. Use that step to make sure you understand what you're being asked to do.

We will be evaluating your docstrings in addition to your code. Please include two examples in your docstrings. You will need to paraphrase the full descriptions of the functions to get an appropriate docstring description.

To understand how these functions might map onto the game, be sure to watch and review the video side by side with thinking of your example function calls:

Functions to write for A1

Function name: (Parameter types) -> Return type	Full Description (paraphrase to get a proper docstring description)
<code>get_section_start</code> <code>(int) -> int</code>	<p><i>Section - length = 3</i></p> <p>The parameter is a section number that corresponds to one of the scrambled string's sections.</p> <pre> xx x, x xx, xxx, xxx 0 3 6 9 0 4 8 </pre>

This function should return the index of the first character in the specified section. For example, with a section length of 4, section 1 begins at index 0, section 2 begins at index 4, section 3 at index 8, and so on. On the other hand, with a section length of 3, section 1 begins at index 0, section 2 begins at index 3, section 3 at index 6, and so on.

`is_valid_move`

`(str) -> bool`

check, swap, rotate

The parameter is a string that may or may not be a valid move in the game.

This function should return `True` if and only if the parameter represents a valid move, i.e. it matches one of the three move constants. if else / or

XXXXXX 3
1 2

`is_valid_section`

`(int) -> bool`

if 3, 4, 5 False

The parameter is an int that may or may not be a valid section number.

This function should return `True` if and only if the parameter represents a section number that is valid for the current answer string and section length. For example, if the answer string is 'wordlockgame' and the section length is 4, then this function should return `True` for the ints 1, 2, and 3, and False for all other ints. input 数字在范围是 0 不行 > < >= <=

用第一个 function

`[:]`

`check_section`

`(str, int) -> bool`

第几个 section

XLXXXXX

Answer: XXXXXX

The first parameter is the game state (i.e. the current state of the scrambled string), and the second parameter is a valid section number.

This function should return `True` if and only if the specified section in the game state matches the same section in the answer string. That is, if the specified section has been correctly unscrambled.

从 XL 的 for answer 取出 section

`change_state`

`(str, int, str) -> str`

section number

XL

取 section [:]

swap / rotate

The first parameter represents the game state, the second parameter is the section number of the section to be changed, and the third parameter is the move to be applied to the string. The move will be one of `SWAP` or `ROTATE`.

This function should return a new string that reflects the updated game state after applying the given move to the specified section. For example, if the section length is 4, and this function is called `change_state('wrdokoclgmae', 2, 'S')`, then the function should return `'wrdolckgmae'`.

if == SWAP

`get_move_hint`

`(str, int) -> str`

In this function only, you may assume a fixed section length of 3.

else:

The first parameter represents a game state, and the second parameter represents the number of a section in the game state that is not yet unscrambled.

This function should return a move (either `SWAP` or `ROTATE`) that will help the player rearrange the specified section correctly.

CTA

rotate 2 3 swap

Use your creativity here, but make sure to give hints that if used in progression, should never produce the same game state twice. For example, always telling the user to play `SWAP` on a section `'ATC'` each round will cause that section of the game state to go back and forth between `'ATC'` and `'CTA'`, and never to get to the answer (`'CAT'`).

If a player repeatedly follows your hints, they should be guaranteed to end up solving the game. Hint: Consider the cases in which `SWAP` is a bad move to make.

Note that this function may not work if `SECTION_LENGTH` is anything other than 3.

Using Constants

As we discuss in section [Constants](#) above, your code should make use of the provided constants. If the value of one of those constants were changed, and your program rerun, your functions should work with those new values.

For example, if `SECTION_LENGTH` were changed, then your functions should work according to the new section length.

Your docstring examples should reflect the given **values** of the constants in the provided starter code, and **do not need to change**.

No Input or Output

Your `wordlock_functions.py` file should contain the starter code, plus the function definitions specified above. `wordlock_functions.py` must *not* include any calls to functions `print` or `input`. Do *not* add any `import` statements. Also, do *not* include any function calls or other code outside of the function definitions.

How should you test whether your code works

First, run the checker and review ALL output — you may need to scroll. You should also test each function individually by writing code to verify your functions in the Python shell. For example, after defining function `change_state`, you might call it from the shell and make sure it does what you expect — copy and paste your example calls, and think about what else might be an interesting test case.

A1 Checker

We are providing a checker module (`a1_checker.py`) that tests two things:

- whether your code follows the [Python Style Guidelines](#), and
- whether your functions are named correctly, have the correct number of parameters, and return the correct types.

To run the checker, open `a1_checker.py` and run it. Note: the checker file should be in the **same** directory as your `wordlock_functions.py`, as provided in the starter code zip file. Here is a video using the checker in last semester's assignment — the process is the same.

[a demo of the checker being run](https://www.youtube.com/watch?v=Y8FZg7GxFdl&feature=youtu.be) [.\(https://www.youtube.com/watch?v=Y8FZg7GxFdl&feature=youtu.be\)](https://www.youtube.com/watch?v=Y8FZg7GxFdl&feature=youtu.be)



<https://www.youtube.com/watch?v=Y8FZg7GxFdl&feature=youtu.be>

Be sure to scroll up to the top of the output and read all messages.

If the checker passes for both style and types:

- Your code follows the style guidelines.
- Your function names, number of parameters, and return types match the assignment specification. **This does not mean that your code works correctly in all situations.** We will run a *different* set of tests on your code once you hand it in, so be sure to thoroughly test your code yourself before submitting.

If the checker fails, carefully read the message provided:

- It may have failed because your code did not follow the style guidelines. Review the error description(s) and fix the code style. Please see the [PyTA documentation](http://www.cs.toronto.edu/~david/pyta/) [.\(http://www.cs.toronto.edu/~david/pyta/\)](http://www.cs.toronto.edu/~david/pyta/) for more information about errors.
- It may have failed because:
 - you are missing one or more function,
 - one or more of your functions is misnamed,
 - one or more of your functions has the incorrect number or type of parameters, or
 - one of more of your function return types does not match the assignment specification.

Read the error message to identify the problematic function, review the function specification in the handout, and fix your code.

Make sure the checker passes before submitting.

Running the checker program on MarkUs

In addition to running the checker program on your own computer, run the checker on [MarkUs](https://markus108.teach.cs.toronto.edu/csc108-2020-01/en/main) [.\(https://markus108.teach.cs.toronto.edu/csc108-2020-01/en/main\)](https://markus108.teach.cs.toronto.edu/csc108-2020-01/en/main) as well. You will be able to run the checker program on MarkUs once every 24 hours. This can help to identify issues such as uploading the incorrect file.

Once you have submitted your work on MarkUs, click on the "Automated Testing" tab and then click on "Run Tests". Wait for a minute or so, then refresh the webpage. Once the tests have finished running, you'll see results for the Style Checker and Type Checker components of the checker program (see both the Automated Testing tab and results files under the Submissions tab). Note that these are not actually marks!

They are only the the checker results. If there are errors, edit your code, run the checker program again on your own machine to check that the problems are resolved, resubmit your assignment on MarkUs, and (if time permits) after the 24 hour period has elapsed, rerun the checker on MarkUs.

Marking

MarkUs is the submission and autotesting system we will use for the term. We have loaded into MarkUs a set of Python programs that, when run, test your code. Specifically, our programs call your functions with various argument values, and verify that the return value is the expected one. The more tests you pass, the higher your mark.

We will not tell you our test cases — verifying that your code works is a fundamental part of computer programming. You should come up with your own test cases.

We also run a test program that does a rudimentary check of your docstrings. Then we run the checker, which uses [PyTA](https://pypi.org/project/python-ta/) [\(https://pypi.org/project/python-ta/\)](https://pypi.org/project/python-ta/), a program that reads your Python files and checks that you are following our [Python Style Guidelines](https://www.python.org/dev/peps/pep-0008/) that we derived from [the PEP8 rules](https://www.python.org/dev/peps/pep-0008/) [\(https://www.python.org/dev/peps/pep-0008/\)](https://www.python.org/dev/peps/pep-0008/).

These are the aspects of your work that may be marked for A1:

- **Coding style (20%):**
 - Make sure that you follow our [Python Style Guidelines](https://www.python.org/dev/peps/pep-0008/) that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we don't provide an exhaustive list of style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with one exception: *docstrings may be evaluated separately*. For each occurrence of a [PyTA](http://www.cs.toronto.edu/~david/pyta/) [\(http://www.cs.toronto.edu/~david/pyta/\)](http://www.cs.toronto.edu/~david/pyta/) error, one mark (out of 20) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.
 - All functions, including helper functions, should have complete docstrings including preconditions when you think they are necessary. Follow the [Python Style Guidelines](https://www.python.org/dev/peps/pep-0008/).
- **Correctness (80%):** Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.

No Remark Requests

No remark requests will be accepted. A syntax error could result in a grade of 0 on the assignment. Before the deadline, you are responsible for running your code and the checker program to identify and resolve any errors that will prevent our tests from running.

Going Further (optional, not for marks)

This assignment is designed to be doable with only the concepts we see in the first few weeks of the course. This means that there are a lot of choices we made in designing this system that could be improved on as we learn more concepts. You are encouraged to think about ways in which you could improve the entire system (both the functions and program files) with additional concepts.

You will notice that there is code provided in `wordlock_game.py` that we have not yet talked about in class. If you are finding this assignment pretty straightforward, we encourage you to figure out how the program code works, and to then think about how to improve it. By the end of this course, you will be able to understand all of the code in the provided functions, and likely even be able to think of ways to improve it! You may also think of better ways to represent the data in the program.

Important: Your code will be evaluated on how it meets the function descriptions above. **While we encourage you to think about how to improve the program, do NOT submit these improvements as part of your Assignment 1 submission.**

What to Hand In

The very last thing you do before submitting should be to run the checker program one last time. Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit your `wordlock_functions.py` file on MarkUs by following the instructions on the course website. Remember that spelling of filenames, including case, counts: your file must be named exactly as above.

You can resubmit as many times as you like up until the deadline. Also, you can get partial marks for each complete and correct function.