

Project 1: Wordle

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Change Log

With a common lab assignment we do not typically have time to change the lab in response to student questions, concerns, and common misunderstandings. However, as the project lasts three weeks, it is relatively common for small updates, extra hints, and corrections to minor typos to be added. This page will list any such modifications.

- Version 1.0 2025-02-11

Contents

1	Change Log	1
2	Essential Information	3
2.1	Pacing and Planning	3
2.2	Learning Goals	3
2.3	Deadlines	3
2.4	Individual Assignment	4
2.5	Appropriate Resources and libraries	4
3	Introduction	6
4	Our End-Goal	7
5	Files and provided functions	8
6	Requirements wordle.py	8
6.1	Required functions	9
6.1.1	check_word	9
6.1.2	known_word	12
6.1.3	no_letters	12
6.1.4	yes_letters	13
7	Runnable Program	13
8	Requirements easy_wordle.py	14
8.1	Required function: filter_word_list	15
8.2	Runnable Program	16
9	Testing	16
9.1	Automatic testing	16
9.2	Manual testing	17
10	Grading	17

2 Essential Information

2.1 Pacing and Planning

This is an approximately 3 week long assignment (Not including Spring Break) As such you should expect it to be longer, and more complicated than past assignments you have undertaken. As this is likely your first large-scale programming project, we will be providing several key decisions for software design, namely some core helper functions, and data representations. There will still be room for you to explore further helpful functions to structure your programming, as well as decisions on how to represent some key pieces of data needed by your algorithms, but, this should be enough to get you started (and importantly allow us a more objective basis for grading.)

Do not underestimate the additional time needed to understand a provided design like this – this is more than a simple reading task – You’re being asked to understand not only a problem, but also one person’s approach to structuring a solution. Make sure you understand how the functions fit into the larger program you need, and the role of each part of this document. If you find yourself thinking “I don’t need this” or “I don’t know what this does, so I’m going to ignore it” there’s a good chance you’re setting yourself up for a painful lesson later when you realize why we designed the problem as we did.

2.2 Learning Goals

A project like this often has many learning goals, both *direct* and *indirect*

- Practice building larger-scale software. Including the time-management skills needed to pace yourself, the organization skills in seeing how each piece of a large program fits into one big picture, and of course, the debugging skills needed for a large task like this.
- Practice designing and implementing many sequence-based operations
- Experience working with larger data than we have in the past, and (therefore) considering efficiency implications.
- Practice structuring larger code in organized ways.
- Practice with larger-scale problem solving
- Practice reading and understanding a larger-scale problem description, and mapping larger problems to specific python ideas.
- (last but not least) Just more practical experience with python

2.3 Deadlines

This assignment will be due Friday March 7th at 6:00pm.

important notes about project-deadlines

- There is no late-work deadline for Project 1. Without prior arrangement, late work will not be accepted.

- There will be a small *Grace period* window put into gradescope to help mitigate technical difficulties during submission. You can submit during the grace period without deduction – if gradescope accepts your work, you will be fine.
- (Of course, we will honor standard umn policy ¹ as it pertains to emergency situations)
- Like with labs, you can make multiple submissions, we will only grade the final submission made before the deadline. If you are getting close to the deadline, it might help you to make a "safety submission" with whatever you have (with the plan to update it as the evening progresses).
- Every semester there are people who fail to finish project 1 not-because they are bad programmers, but because they do not dedicate enough time. Do not be one of these people. Plan to be finished several days before the deadline, and aim to make incremental progress. This is a marathon, not a sprint. Give the project the time it deserves and you will learn much more from the experience.

2.4 Individual Assignment

Unlike labs, where partner work is allowed, this project is an *individual assignment*. This means that you are expected to solve this problem independently relying only on course resources (zybook, lecture, office hours) for assistance. Inappropriate online resources, or collaboration at any level with another student will result in a grade of 0 on this assignment, even if appropriate attribution is given. Inappropriate online resources, or collaboration at any level with another student without attribution will be treated as an incident of academic dishonesty. Like on homework – you can discuss *what the PDF is asking you to do* but you should not collaborate on problem-solving approaches to the problem. If you are unclear if a resource is reasonable, or if communication will be allowed, please seek clarification from course staff.

2.5 Appropriate Resources and libraries

You are expected to be careful to only use appropriate resources in this work. While relatively new (in the broad scheme of things) Wordle was wildly popular for a period of time, and as such, there have been many different attempts and approaches at making clones and assistive tools. **You should not be referencing these solutions**, many of them will not help you with this project, but more importantly **The point of the project is to practice problem-solving** you cannot do this if you don't solve the problem yourself. "Looking up the solution" is not a problem solving technique that you can rely on in practical situations.

You are, of course, free to use online resources to learn about *python itself*. Likewise, it's entirely appropriate to use online resources to better understand the game Wordle itself. You are, however, forbidden from using outside resources to learn how to solve or simulate Wordle itself. All we're asking is that the approaches you take to solve this problem are your own design – creating approaches to problems like this is the skill you build by doing large

¹<https://policy.umn.edu/education/makeupwork>

projects. If you do use any resources that have not been discussed in class, you are expected to leave a citation about these resources using a python comment in your code. This will allow us to know what resources you've used, especially if we see something unexpected in your code. Failure to do so may be viewed as an academic integrity issue.

Additionally, please remember the “common libraries” rule from class: You can only use python libraries we've explicitly taught in class, such as math or random. Other libraries require you to explicitly seek permission. Use of other libraries without permission can lead to a grade of 0.

3 Introduction

The game Wordle has a short and interesting history, originally designed in 2013 as a letter-based copy of the color-matching game Mastermind, Wordle quickly became popular during the Covid-19 lock downs of 2020 and 2021. As of now, Wordle is owned by the New York Times, and can be played for free online at <https://www.nytimes.com/games/wordle/index.html> (If you have not played wordle I recommend following this link and playing it now. It's easier to learn interactively than by reading a description)

The game of Wordle is simple to play. In each game a 5 letter English word is chosen. The player gets six attempts to guess this five letter word. Each guess is limited to be a real word (I.E. the game only accepts real words as guesses, typing not-a-word will not cost a guess) After each guess the player is given a clue as feedback. This clue will present the players last guess with colors annotating each letter

1. Green – a letter is green to indicate the secret word contains this letter at this position
2. Yellow – a letter is yellow to indicate that the secret word contains this letter, but not at this position
3. Grey – a letter is *typically* grey to indicate that the word does not contain this letter.

For a simple example, consider the following clue we could get back after guessing "audio"



From this we can conclude:

- The fourth letter is I
- The letter A appears at least once in the word.
- The first letter is not A
- U, D, and O do not appear in the word.

Making this simple idea more complicated, is the fact that, in the English language, some words contain duplicate letters. For instance, the word "books" contains two "o"s. In this case the rules for clues become more complicated, as additional letters will be colored yellow or green only if the letter is also repeated in the secret word. We'll look at an example, and then see more detail after. For example consider the guess "LLAMA" (which has two L and two A)



Things to note:

- The first L is yellow, and the second is green – this tells us that there are AT LEAST 2 L in the word, that the first letter is not L, and the second letter is L.
- One A is green, and the other is grey. This tells us that there is EXACTLY 1 'A' in the word, and that it's the third letter.
- Letters can be grey (the second A) without necessarily indicating the letter is not in the word.

- For more information I'll direct you to various online resources explaining the phenomenon – this website was pretty clear in my opinion: <https://nerdschalk.com/wordle-same-letter-twice-rules-explained-how-does-it-work/>

4 Our End-Goal

We will ultimately be building two Wordle clones. The first Wordle clone will have the same amount of support as standard Wordle: it will keep track of which spaces in the word are known, which letters must be in the word, and which must not be in the word. The second Wordle clone will be "Easy mode" – and will tell you how many possible words are left, and provide you with a sample of possible words after each guess. Screenshots (to preserve color) can be seen below for these games. In each case, only the text after the > was typed – the rest was computer output. Normal Wordle is on the left, easy wordle is on the right.

```
Known:  -----
Green/Yellow Letters:
Grey Letters:
> radio
RADIO
Known:  -----
Green/Yellow Letters: DR
Grey Letters: AIO
> druid
RADIO
DRUID
Known:  ____D
Green/Yellow Letters: DR
Grey Letters: AIOU
> lered
RADIO
DRUID
LERED
Answer: LERED
```

```
> raise
RAISE
39 words possible:
siver
hires
icers
fires
tires
> serif
RAISE
SERIF
2 words possible:
meris
peris
> meris
RAISE
SERIF
MERIS
1 words possible:
peris
> peris
RAISE
SERIF
MERIS
PERIS
1 words possible:
peris
```

5 Files and provided functions

Like many projects, this program will involve multiple files connected through python `import` statements. Some of these files will be pure python modules – containing only function and variable definitions. Others will also include a python program (guarded by `if __name__ == "__main__":` so functions in the file can still be imported)

The files for this project:

- `display_utility.py` (PROVIDED) This file contains methods designed to use ANSI escape sequences to allow us to have colors in our terminal output. These escape sequences are *kind of archaic* so don't be surprised if you can't figure out exactly how the code *works*. So long as you can use the provided docstrings and examples to understand how to use this code you should be fine. (If you have trouble with this module contact course staff, we can help debug, or provide an alternative)
- `words.py` (PROVIDED) This file contains around 15,000 5 letter words. This is taken from an online github repository (see citation in the words file itself) which *claims* that this is the entire Wordle word-list. While it's possible this word list isn't exactly the same as the official one, it's good enough for now. **IMPORTANT NOTE** this file is pretty big, and may upset some computers. It only has one variable `words`. You may want to skip opening this file and just use `from words import words` to import the word list into your code. Each word is stored lowercase – you will need to uppercase it with the `x.upper()` string method.
- `wordle.py` (YOU WRITE) This will contain a series of functions that implement core behaviors for playing Wordle, as well as core tools for making logical deductions from Wordle clues. This file will also contain a program to play Wordle.
- `easy_wordle.py` (YOU WRITE) This file should be much shorter than `wordle.py`. It will have one required function, and the "easy mode" wordle.

6 Requirements wordle.py

This file will contain both functions (Required, as well as helper functions of your own design) and an interactive program. Since you will need to import functions from this file for the next file, this file should have the "standard python structure" we discussed in lecture.

```
import statements

functions
functions
functions
functions

if __name__ == "__main__":
    user interface code
    user interface code
```


6.1 Required functions

While you will probably want to structure your program into more than just these functions, we are only requiring the following:

- `def check_word(secret, guess)`
- `def known_word(clues)`
- `def no_letters(clues)`
- `def yes_letters(clues)`

The inputs for the above are as follows (listed here as there is meaningful overlap):

- `secret` – a string, the secret word. You can assume this is all uppercase letters, and 5 letters long.
- `guess` – a string, the guessed word. You can assume this is all uppercase letters and 5 letters long.
- `clues` – a list of tuples, with each tuple being a guess (string) and the clues returned (a list of strings, see `check_word`). This will look like the following:

```
[(guess, clue), (guess2, clue2), (guess3, clue3), ...]
```

6.1.1 `check_word`

The check word function is in charge of taking a word and generating the output clue. The input is the secret word for the Wordle game, and the guess input by the user. You should assume that the user-interface code already handled validation and formatting here – so the inputs are length 5 strings formed of only uppercase letters, and that they are in the words list. The output will be a length-5 list containing the strings "grey", "yellow", and "green".

The rules for clue generation are described above, for simple words without repeats this is not complicated – "green" in position `i` of the output list means that the letters in position `i` of the secret and guess are the same. "yellow" in position `i` means that the letter `guess[i]` is in `secret` but is not `secret[i]`, and "grey" in position `i` means that the letter `guess[i]` does not appear in `secret`.







Unfortunately, things get more complicated when you account for words with double-letters (which do happen, so we must handle then, as was mentioned in the introduction section). If we applied the "basic" rules above we could get into some very tricky situations where clues could be misleading, therefore Wordle has more intricate rules for this situation. The best way to understand this is as a *series* of rules that have to be applied in order

- Green rule: Any letter in the guess that is correct should get marked green (as before)
- Yellow rule: moving left-to-right over the guess, mark a letter yellow if:
 - The guessed letter is not already marked green (i.e. the position is wrong)
 - The guessed letter does occur in the secret word
 - The matching letter in the secret word has not been used to give a green or yellow clue already.
- Grey rule: if we cannot assign a green or yellow clue assign a grey clue.

The effect of these rules is that we only hand out as many yellow or green clues as matching letters in the guess, so if we had secret word "octal" but guessed "books" only one of the

"o" in books would be yellow (since only one "o" appears in "octal") However, if we guessed "books" for the secret word "outgo" then both "o" in books could be yellow since "o" occurs twice to in "outgo" so there's a match for both "o". Since these rules can be a bit confusing if you only read them, consider the following examples.

If you're having trouble with these rules make sure you step through these examples in depth and think about how the inputs relate to the correct outputs (This might sound like "busywork" but this sort of reflection and inspection is a core part of the problem-solving process for most programmers.)

Secret	Guess	output
OPENS	EPICS	["yellow", "green", "grey", "grey", "green"]  A pretty straightforward example. P and S are in the right place and are hinted green. E is in the wrong place, but in the OPENS so it is hinted yellow. I and C are not in OPENS and are grey.
MISTS	MISTY	["green", "green", "green", "green", "grey"]  Another pretty straightforward example. Note that no indication is given that the secret has two S. Also note that the S in Misty is green, not yellow (both technically apply)
LERED	DRUID	["grey", "yellow", "grey", "grey", "green"]  LERED only has one D, so only one D in DRUID should be hinted. The green second-D takes priority over making the first D yellow.
ELUDE	LEDGE	["yellow", "yellow", "yellow", "grey", "green"]  Elude and Ledge both have two E, so both E in LEDGE have hints – the second one matches so it's green.
CRANE	BEEPS	["grey", "yellow", "grey", "grey", "grey"]  Crane only has one E so only one of the E in BEEPS is hinted. Both are in the wrong place, so the hint is yellow, and we hint the first E by default.
ROBOT	REORG	["green", "grey", "yellow", "grey", "grey"]  Another good example of double-letter behavior. both the double R in Reorg (only hinted once as Robot has only one R) and the double-O in Robot (only hinted once because Reorg has only one O)

Some notes/hints on implementation:

- This function is easy to do poorly, and hard to do well.
 - Make sure to test your function carefully, use a broad range of examples, including examples which have "double letter" situations.
 - Many students approach this by first implementing the "basic" rules are trying to add the more complicated version of the rules in later. This often leads to

throwing your first-draft code away. **We recommend planning your code to handle the complicated double-letter situations correctly from the start.** This will help avoid needless work.

- As indicated by the multi-step part of the rules, you might need to handle this as a multi-step process, rather than trying to do this in one loop.
- This is not one of those problems with only one "good" solution – there are many valid ways to solve the problem (and plenty of other invalid ones...) As such, don't get too stuck looking for the "golden solution" – focus instead of coming up with a solution that makes sense to you, and you can be confident provides the right results.
- Most algorithms we can think of require *extra data* to be stored as you progress (for instance, you need some way to remember if a letter has been used for a green-clue to avoid re-using it for a yellow clue!) It will be up to you to decide what information your algorithm needs to remember as it goes, and how best to remember that.
- Don't let the difficulty of understanding the wordl rules, or designing an algorithm intimidate you – this is part of the programming process. Understanding intricate tasks and designing tricky algorithms is part of the programming process. With some practice this will come more naturally.
- If you're finding this difficult, you will find it useful to work through examples by-hand with pencil/paper – the clue-assignment strategy is more complicated than it appears, and doing it yourself can help you resolve that complexity.

6.1.2 known_word

The input to this function is a list containing a record of guesses taken and clues received so-far. The format is seen at the top of section 6.1 The output of this function is a String indicating what we know about the secret word according to green hints seen so-far. The output of this can be seen as the "Known" string in the normal wordle example in section 4: `__ING` where positions which have not seen a green clue are presented as `_` and positions which have received a green clue are the known letter.

We will use this function to help with "basic recordkeeping" for the end-user of our code, but you could also imagine using this function as part of a Wordle solver.

NOTE in theory it's possible to know the position of a letter without getting a green clue – this can happen if you try all 4 other positions for a letter getting a yellow clue each time. In such a case we do, in theory, know the position of the letter. Regardless, this function would not expected to indicate the position of that letter. All we're going to do in this function is keep track of which positions have seen green clues.

6.1.3 no_letters

The input to this function is a list containing a record of guesses taken and clues received so-far. The format is seen at the top of section 6.1 The output of this function is a String indicating which letters we know are not in the word according to grey hints seen so-far. The output of this can be seen as the "Grey Letters" string in the normal Wordle example

in section 4.

We will use this function to help with "basic recordkeeping" for the end-user of our code, but you could also imagine using this function as part of a Wordle solver.

A few notes:

- The output should only show each letter once no matter how many times it has been guessed.
- The output should be in alphabetic order.
- The output should be all caps
- The output **should not** include a letter that is grey, but is in the word (which can happen in double-letter situations.) Only include a letter in this output string if each occurrence of it in a guess is indicated grey.

6.1.4 yes_letters

The input to this function is a list containing a record of guesses taken and clues received so-far. The format is seen at the top of section 6.1 The output of this function is a String indicating what letters are in the word based on yellow and green hints we've seen so-far. The output of this can be seen as the "Green/Yellow Letters" string in the normal Wordle example in section 4.

We will use this function to help with "basic recordkeeping" for the end-user of our code, but you could also imagine using this function as part of a Wordle solver.

A few notes:

- The output should only show each letter once no matter how many times it has been guessed.
- The output should be in alphabetic order.
- The output should be all caps

7 Runnable Program

The worldle.py file should also implement a runnable program. The `if __name__ == "__main__":` must be used so that this program only runs if the file is executed directly (as you will need to import functions from this file for the next required file)

This program should implement a playable Wordle game. It should start by randomly selecting a word from the provided words list in secret. Then the user should get 6 guesses at this hidden word. Before each guess the current known-letters, green/yellow letters, and grey letters (as defined by the functions above) should be printed. After each guess, a complete record of the guesses and clues seen so far should be printed. You will need to use the functions in `display_utility.py` to typeset this.

When guessing – please only use `>` (greater-than + space) as a prompt. This prompt (and only this prompt) should be repeated until a valid guess is entered. A guess is invalid if: it is not in the words list, or it is not 5 letters long. The words list is stored in lower-case, you are free to require all guesses to be lower-case as well.

Importantly, if a guess is not valid, an appropriate error message should be printed and the prompt should be repeated; but, it shouldn't count as one of the user's 6 guesses. An example can be seen below:

```
Known: _____
Green/Yellow Letters:
Grey Letters:
> aeiou
Invalid word
> aaaaa
Invalid word
> darnit
Word must be 5 letters long
> banana
Word must be 5 letters long
> cat
Word must be 5 letters long
> robot
ROBOT
Known: _0___
Green/Yellow Letters: 0
Grey Letters: BRT
> beees
Invalid word
> roads
ROBOT
ROADS
Known: _0___
Green/Yellow Letters: 0
Grey Letters: ABDRST
```

If the correct word is guessed, or all 6 guesses have been used – the secret word should be shown, and the program should end. There is no difference in formatting between the two cases.

We will be manually grading this part of the assignment only. That said, the expectation is that you do your best to match these exact formatting and output displays. A meaningfully different program behavior (even to the level of extra print statements) will be seen as an incorrect output. As such, please reference the examples in this PDF and match our output format. I understand you may wish to improve on the interface seen in this PDF, but for a matter of grading we will ask for you to simply try to match what we did.

8 Requirements easy_wordle.py

While designing this program we found that our version of Wordle felt a little harder than the official one. Our current theory is that our word list might be a bit larger, making it a bit harder to know what words might be possible. As such, we invented "easy Wordle". In easy Wordle, rather than showing you the normal clues (known letters, letters definitely in the word, and letters definitely not in the word) we simply show a count of how many words

match the given clues, and provide a selection of those words. This file has two parts:

8.1 Required function: `filter_word_list`

The function `filter_word_list(words, clues)` takes a list of words and the clue list format described in section 6.1. It returns a new word list containing only the words in the input word list which could be the secret word.

It might sound hard to know if a given word in the words list *could be* the secret word. Clearly there are some "logic based" solutions that could cover the easy cases, but making something that works perfectly may sound tricky. Fortunately, for a computer, an easier opportunity presents itself, known as a "brute-force" approach. Instead of trying to logic out if a word *could* generate all the given clues we've seen so far, we can simply use the `check_word` we wrote earlier and see if a word *does* generate those clues (when used in-place of the secret word). So for example, given the clues:

A	P	P	L	E
L	L	A	M	A

we could check the word "class" like so

- `check_word("class", "apple")= ["yellow", "grey", "grey", "yellow", "grey"]`
- `check_word("class", "llama")= ["grey", "green", "green", "grey", "grey"]`

While "class" can generate the same clue we saw for guess "apple", it doesn't generate the same clue for guess "llama" so we know it can't be the secret word.

Alternatively could check the word "flail":

- `check_word("flail", "apple")= ["yellow", "grey", "grey", "yellow", "grey"]`
- `check_word("flail", "llama")= ["yellow", "green", "green", "grey", "grey"]`

In this case the clues match for both guesses we've seen so far, so flail could be the secret word.

Some notes

- This function should not modify the word list given to it, or the clue list given to it.
- You *should not* assume the word list given to this function is the original word list from `words.py`. It may be another word list.
- As words lists are typically stored in lowercase, but all other functions assume capitalized inputs, you will need to convert words to uppercase in this function using the `word.upper()` method.
- If you don't understand the algorithm above – contact class staff – trying to come up with another way to do this problem correctly will likely result in an incorrect answer, or a very complicated piece of code (neither of which are worth your time to program!)

8.2 Runnable Program

The `easy_wordle.py` file should also implement a runnable program. The `if __name__ == "__main__":` must be used so that this program only runs if the file is executed directly.

This program should implement a playable Wordle game, following the same basic structure as the `wordle.py` file. You can see an example of this program in section 4. Essentially this is the same game, however the "known", "Green/Yellow Letters" and "Grey Letters" clues are gone. Instead **after** each accepted guess (and its clues) the number of possible words is shown, and up to 5 random possible words.

Notes/details

- If fewer than 5 words are possible, all words should be shown. These can be in any order.
- If more than 5 words are possible only 5 of them should be shown. Choose these 5 at random
- The same word should not be displayed twice. There are many ways to do this, but an easy one is to shuffle the words and print the first 5.
- Don't forget that you can look up what functions are in the random module – there are a few that could be very useful here. Make sure you read more than just their name, some of these functions may not work the way you guess.
- While we normally discourage copy/pasting code from one part of your code to another (rather than using functions to support code-re-use) we will overlook it in this case. While I expect you to look for useful helper functions you can design and implement to make the runnable programs easier to write, and reduce how much code is shared between the two versions, I also understand that it might be easiest to write `wordle.py`'s program then just copy it into `easy_wordle.py` and edit as needed.
- Like with the `wordle.py` function, we may attempt to design some automated tests for this code, as such we ask you attempt to match the prompts seen in the examples letter-to-letter.

9 Testing

Testing will take two separate forms: automatic testing, and manual testing:

9.1 Automatic testing

Automatic testing can be done on your computer using the `test_wordle_funcs.py` file that is posted to canvas. This follows the normal tester file format – you should be able to run it and get the exact listed outputs. When you submit your code on gradescope we will run essentially the same tests in an automated framework for grading. This is *mostly* (but not 100%) the examples from the PDF.

9.2 Manual testing

Manual testing will be done by downloading your code and running it. We will primarily focus on the runnable python programs and be visually making sure they seem to follow the required behaviors. Please do our graders a favor and make your runnable python programs as close as possible in input/output format as shown in the examples. Since we're doing manual review for this – you have *some* flexibility in input/output format, but if you stray too far from this format you might lose points due to us mistaking what the output means.

10 Grading

The approximate grading rubric is as follows

- 33 points autograder for required functions
- 4 "all or nothing" points for behaving correctly when we manually run your wordle program
- 3 "all or nothing" points for behaving correctly when we manually run your easy_wordle program
- 10 points Code Style (same guidelines as applied in python labs, check those documents for details)
- 50 points manual code review:
 - 14 points manual review of wordle runnable program
 - 8 points manual review of wordle runnable program
 - 6 points check word
 - 4 points known word
 - 4 points yes letters
 - 6 points no letters
 - 8 filter word list