CSC110 Fall 2022 Assignment 2: Logic, Constraints, and Wordle!

Print this handout

In the past few weeks, we've gone from writing simple expressions in Python to writing complex functions involving comprehensions, if statements, and tabular data. We've learned about predicate logic, a symbolic mathematical language to express boolean statements precisely and unambiguously, and have gotten lots of practice in translating between English, predicate logic, and the Python programming language. We've even learned about how to write some proofs! On this assignment, you'll put all of what you've learned to use to reason about and simplify if statements, understand and prove mathematical statements in symbolic logic, and model a new problem domain—Wordle!—using a set of definitions that translate naturally into Python functions.

Advice for Assignment 2

We know you have your first term test coming up, so you might not want to get started on this assignment right away! However, we *strongly recommend* taking a few minutes to do the "First impressions" tasks we described on Assignment 1 (../../a1/handout/):

1. Skim the assignment handout.
2. Download the starter files from MarkUs.
3. Schedule time to work on the assignment.

This assignment covers (roughly) Chapters 3 and 4 of the Course Notes, and of course builds on the material in Chapters 1 and 2 as well. That means that reading through the handout—and even starting to work on the first two Parts, which are more accessible than Part 3—will cause you to do naturally review knowledge and skills that will be asses on the term test.

Overall, this assignment involves less written work than Assignment 1, with some deeper analysis and more complex programming tasks. However, we've taken the same approach of breaking down the assignment into multiple parts (and sub-parts) to make it easier for you to process everything, and make a plan to complete the assignment in chunks.

One final note: Part 3 has a lot of reading, which can be intimidating at first! This is fairly normal when introducing a new problem domain (Wordle!), so it's something that you'll get used to as you continue to work on assignments in your computer science courses. Our advice here is to break down the text into *definitions* (introducing key terminology), *examples* (illustrating the meaning of definitions), and *instructions* (what you actually need to do).

Logistics

- Due date: Wednesday, October 12 before 12pm noon Eastern Time.
    - We are using **grace credits** to allow you to submit your work up to 30 hours after the due date. Please review the Policies and Guidelines: Assignments page (https://q.utoronto.ca/courses/278340/pages/policies-and-guidelines-assignments?module_item_id=3928858) carefully!
- This assignment must be done **individually**.
- You will submit your assignment solutions on MarkUs (see Submission instructions at the end of this handout).

Starter files

To obtain the starter files for this assignment:

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09) and go to CSC110.
2. Under the list of assignments, click on **Assignment 2**.
3. On the assignment page, click on the **Download Starter Files** button (near the bottom of the page). This will download a zip file to your computer.
4. Extract the contents of this zip file into your `csc110/assignments/` folder.
5. You should see a new `a2` folder that contains the assignment's starter files!

You can then see these files in your PyCharm `csc110` project, and also upload the `a2.tex` file to Overleaf to complete your written work.

General instructions

This assignment contains a mixture of both written and programming questions. All of your written work should be completed in the `a2.tex` starter file using the LaTeX typesetting language. You went through the process of getting started with LaTeX in the Software Installation Guide (https://q.utoronto.ca/courses/160038/pages/setting-up-your-computer-start-here?module_item_id=1346385), but for a quick start we recommend using the online platform Overleaf (https://www.overleaf.com/) for LaTeX. Overleaf also provides many tutorials (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to help you get started with LaTeX.

Tip: See this video tutorial (https://www.overleaf.com/learn/how-to/How_to_upload_files_to_a_new_project) for instructions on how to upload a `.tex` file to Overleaf.

Your programming work should be completed in the different starter files provided (each part has its own starter file). We have provided code at the bottom of each file for running doctest examples and PythonTA on each file. We are **not** grading doctests on this assignment, but encourage you to add some as a way to understand and test each function we've asked you to complete. We *are* using PythonTA to grade your work, so please run that on every Python file you submit using the code we've provided.

Warning: one of the purposes of this assignment is to evaluate your understanding and mastery of the concepts that we have covered so far. So on this assignment, you may only use parts of the Python programming language that we have covered in Chapters 1-4 and Section 5.1 of the Course Notes. Other parts (e.g., ternary ifs, data classes, loops) are not allowed, and parts of your submissions that use them may receive a grade as low as **zero** for doing so.

Part 1: Conditional Execution

Your work for this part should be done in two Python files: `a2_part1_q1_q2.py` (Questions 1 and 2) and `a1_part1_q3.py` (Question 3).

You may find it helpful to review Section 3.5 of the Course Notes (../../notes/03-logic/05-simplifying-if-statements.html) for this part.

1. Each of the following function bodies contains at least one *nested* if statement (i.e., an if statement within a branch of another if statement).

   In `a1_part2_q1_q2.py`, write a new version of each function so that its body consists of *just one single if statement* (possibly with multiple `elif` branches).

   Requirements:

   - All return statements in the function body must be contained within branches of the if statement (including possibly the `else` branch).

- All boolean expressions must be simplified to remove redundant conditions (e.g., `x > 1 and x > 0` should be simplified to `x > 1`).

a.
```python
def mystery_1a_nested(x: list[int], y: list[int]) -> list[int]:
    """Mystery 1a."""
    if len(x) > 1 and len(y) > 1:
        return x
    else:
        if sum(x) > sum(y):
            return y + x
        else:
            return x + y
```

b.
```python
def mystery_1b_nested(n: int, nums: set[int]) -> int:
    """Mystery 1b."""
    if n < len(nums):
        if n == 1:
            return 0
        else:
            if n % 2 == 0:
                return sum(nums)
            else:
                return sum(nums) + n
    else:
        return len(nums)
```

Hint: simplify from the inside-out, starting with the most deeply nested if statement.

2. Each of the following function bodies contains at least one if statement. In `a1_part2_q1_q2.py`, write a new version of each function so that its body does not use *any* if statements.

   We encourage you to use local variables to save intermediate values when computing a larger boolean expression. Like Question 1, all boolean expressions must be simplified to remove redundant conditions (e.g., `x > 1 and x > 0` can be simplified to `x > 1`).

   Hints:

   - You might find it useful to approach this problem by identifying exactly what specific conditions make the function return `True` (or, if easier, what specific condition(s) make the function return `False`).
   - You might find it useful to start by simplifying each nested if statement separately first.

a.
```python
def mystery_2a_if(x: int, y: int) -> bool:
    """Mystery 2a."""
    if x < y:
        if 2 * x < y:
            return True
        elif 2 * x > y:
            return False
        else:
            return False
    else:
        if x == y:
            return False
        elif 2 * y < x:
            return False
        elif 2 * y > x:
            return True
        else:
            return False
```

b.
```python
def mystery_2b_if(x: int, y: int) -> bool:
    """Mystery 2b."""
    if x >= 0:
        if y >= 0:
            if x >= y:
                return False
            else:
                return True
        else:
            return True
    else:
        return True
```

3. Consider the following function:

```
    def mystery_3(s1: str, s2: str, s3: str) -> int:
        """Mystery 3.
        """
        if len(s1) != len(s2) or len(s1) != len(s3):  # Branch 1
            return 1
        elif s1 == '':                                 # Branch 2
            return 2
        elif s1 == s2 + s3:                            # Branch 3
            return 3
        else:                                          # Branch 4
            return 4
```

This function has an if statement with four different branches. However, not all four branches are *reachable*: for at least one of the branches, its return statement will *never* be executed for any valid input. Let's explore!

a. In `a2_part1_q3.py`, complete the set of *unit tests* for this function so that together, the tests cover all possible *reachable branches* in this function body. Essentially, what you are doing is: for each reachable branch, showing that *there exists* inputs `s1, s2, s3` to the function such that `mystery_3(s1, s2, s3)` causes the code in that branch to execute.

b. In `a2_part1_q3.py`, write a set of *property-based tests* for this function so that for each *unreachable branch* in the function body, there is a test that asserts that the corresponding return value is never actually returned. For example, if you believe that Branch 1 is unreachable, you should have a test which asserts

```
assert mystery_3(s1, s2, s3) != 1
```

c. *Not to be handed in, but interesting to think about!* For each unreachable branch in `mystery_3`, **prove** that it is unreachable: for all strings `s1, s2, s3`, calling `mystery_3(s1, s2, s3)` will not execute that branch.

Part 2: Proof and Algorithms, Greatest Common Divisor edition

Note: this part is a written response, and should be completed entirely in `a2.tex`.

Let's start with a definition.

Let $x, y, d \in \mathbb{Z}$. We say that d is the **greatest common divisor** of x and y when it is the largest number that is a common divisor of x and y, *or* when $d = 0$ if x and y are both 0. We define the function $\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ as the function which takes numbers x and y and returns their greatest common divisor.

For example, $\gcd(10, 4) = 2$ and $\gcd(0, 0) = 0$.

Consider the following function for calculating the greatest common divisor of two *positive* integers:

```python
def gcd(n: int, m: int) -> int:
    """Return the greatest common divisor of m and n.

    Preconditions:
    - 1 <= m <= n
    """
    possible_divisors = range(1, m + 1)
    common_divisors = {d for d in possible_divisors if divides(d, n) and
        divides(d, m)}
    return max(common_divisors)


def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
    if d == 0:
        return n == 0
    else:
        return n % d == 0
```

1. David asks, "Why does this approach use `range(1, m + 1)` instead of `range(1, n + 1)`?" Use one of the properties of divisibility we stated in **Lecture 9** to answer this question.

2. Tom asks, "We know that calling `max` on an empty collection raises an error. Why aren't we checking for this case before calling `max(common_divisors)`?" Use one of the properties of divisibility we stated in **Lecture 9** to answer this question.

3. Recall the following definition (adapted from the *Quotient-Remainder Theorem*):

Let $n, d \in \mathbb{Z}$ with $d \neq 0$. We define the **remainder** of n divided by d to be the unique integer r that satisfies both:

1. $0 \leq r < |d|$, and
2. $\exists q \in \mathbb{Z},\ n = qd + r$

We use the notation $n \% d$ (or in Python, `n % d`) to denote the remainder of n divided by d.

Prove the following statement:

$$\forall n, m, d \in \mathbb{Z},\ d \mid m \land m \neq 0 \Rightarrow \big(d \mid n \Leftrightarrow d \mid n \,\%\, m\big)$$

Notes:

- Please make sure to set up your proof with the same structure we saw in lecture.
- To prove an "if and only if" $p \Leftrightarrow q$, you should do this in two parts: first prove that $p \Rightarrow q$, and then prove that $q \Rightarrow p$. We have provided some examples of this type of proof structure in the Course Notes (Sections 4.6/4.7).
- Use the following property of divisibility (without proving it): $\forall n, m, d, a, b \in \mathbb{Z},\ d \mid n \land d \mid m \Rightarrow d \mid (an + bm)$. Clearly state how/where you are using it in your proof, referring to it as ``the given property" or something similar.

4. Using the statement from Question 3, and one or more of the properties of divisibility we stated in **Lecture 9**, show how we can modify the `gcd` implementation from above to check for common divisors across a range of numbers *smaller* than `range(1, m + 1)`.

   Please do two things:

   a. Fill in the following updated implementation of `gcd` (found in `a2.tex`):

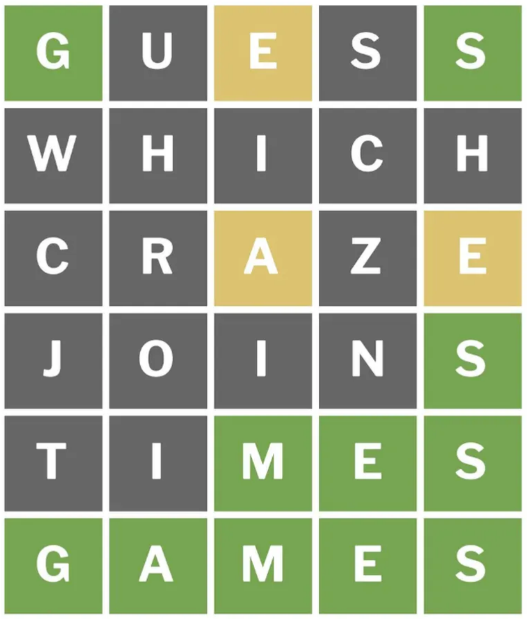
```python
def gcd(n: int, m: int) -> int:
    """Return the greatest common divisor of m and n.


    Preconditions:
    - 1 <= m <= n
    """
    r = n % m  # (Sept 30) CORRECTED: Used to be n % d


    if r == 0:
        ...
    else:
        possible_divisors = ...
        common_divisors = {d for d in possible_divisors if divides(d, n)
         and divides(d, m)}
        return max(common_divisors)
```

Note: even though you're submitting this code in PDF form, we strongly encourage you to test it yourself before submitting it!

b. Provide an English explanation of the changes you made, and why you made them. A formal proof is not necessary.

Part 3: Wordle!

You might have heard of **Wordle**, an online word guessing game that went viral at the end of 2021. In this game, there is a secret "answer" word, and the goal is to guess that word in at most six guesses. After each guess, you get information about whether each letter in your guess word appears in the answer word: a green letter means that the letter is in the answer word at that position, an orange letter means that letter is in the answer word in a different position, and a grey letter means that the letter does not appear in the answer word.



If you haven't played Wordle before, we strongly recommend playing a few games on the official Wordle website ((https://www.nytimes.com/games/wordle/index.html)) or Wordle Unlimited (https://wordleunlimited.org/), the latter of which lets you play as many games as you like per day.

In this part of this assignment, you'll implement a program that models the Wordle game using ideas from mathematical logic. That might sound a bit strange at first, but by modelling the "problem domain" of Wordle using logic, we'll be able to create a program that can solve Wordle puzzles automatically, and even work backwards to recreate Wordle boards from a given answer. Let's get started!

Representing Wordle's data

Before we get to any programming, we need to better understand the problem domain. As you might expect, the best way to start this is with some definitions.

- A **word set** is a collection of all valid words in a Wordle game. In this section, we'll use the variable W to represent the word set.
  - *Note*: in the official Wordle game, all words have exactly five letters. We'll use a more general constraint for this assignment: all words in the word set must have the same non-zero length. For example, we might have a word set containing all six-letter English words.
- An **answer** $a \in W$ is the word that the Wordle player is trying to guess.
- A **guess** $g \in W$ is a word that the Wordle player has chosen as a guess.[1]
- Let $a \in W$ be an answer word, $g \in W$ be a guess word, and $i \in \mathbb{N}$ be a valid index for these words (i.e., $i < len(a)$). The **character status for guess g at index i with respect to the answer** a is defined to be one of three possible values:
  - **CORRECT**, when $g[i] = a[i]$
  - **WRONG_POSITION**, when $g[i] \neq a[i] \land \big(\exists j \in \mathbb{N},\ j < len(g) \land j \neq i \land g[i] = a[j] \land g[j] \neq a[j]\big)$.

    Intuitively, this means that the letter $g[i]$ is not correct, but appears at a different index j in the answer word a, *and* $g[j]$ isn't already correct.

    Note: The last condition may seem a bit strange, but it allows us to handle duplicated letters within words, though in a simpler way than the official Wordle game. *See examples below.*
  - **INCORRECT**, when the status is not CORRECT or WRONG_POSITION.

  These character statuses correspond to green, orange, and grey colours in the official Wordle game, respectively.
- Let $a \in W$ be an answer word and $g \in W$ be a guess word. The **guess status of g with respect to** a is the list of character statuses for each character in g.

Example (Wordle definitions: guess, answer, status)

Suppose our answer word is $a =$ 'hello'.

1. If our guess word is $g =$ 'reach', the character status for each index is:

Index i	$g[i]$	$a[i]$	**Status**
0	‘r’	‘h’	INCORRECT
1	‘e’	‘e’	CORRECT
2	‘a’	‘l’	INCORRECT
3	‘c’	‘l’	INCORRECT
4	‘h’	‘o’	WRONG_POSITION (taking $j = 0$ in the definition of WRONG_POSITION)

2. If our guess word is $g =$ ‘allow’, the character status for each index is:

Index i	$g[i]$	$a[i]$	**Status**
0	‘a’	‘h’	INCORRECT
1	‘l’	‘e’	WRONG_POSITION (taking $j = 3$)
2	‘l’	‘l’	CORRECT
3	‘o’	‘l’	WRONG_POSITION (taking $j = 4$)
4	‘w’	‘o’	INCORRECT

3. If our guess word is $g =$ ‘keeps’, the character status for each index is:

Index i	$g[i]$	$a[i]$	**Status**
0	‘k’	‘h’	INCORRECT
1	‘e’	‘e’	CORRECT
2	‘e’	‘l’	**INCORRECT**
3	‘p’	‘l’	INCORRECT
4	‘s’	‘o’	INCORRECT

Note that the second ‘e’ in ‘keeps’ (at index 2), is considered *INCORRECT*, not WRONG_POSITION, even though the letter ‘e’ appears in the answer word. This is because ‘e’ already appears at the same index in the guess word (i.e., $g[1] = a[1]$).

4. Finally, if our guess word is $g =$ ‘hoops’, the character status for each index is:

Index i	$g[i]$	$a[i]$	**Status**
0	‘h’	‘h’	CORRECT

Index i	$g[i]$	$a[i]$	**Status**
1	'o'	'e'	**WRONG_POSITION** (taking $j = 4$)
2	'o'	'l'	**WRONG_POSITION** (taking $j = 4$)
3	'p'	'l'	INCORRECT
4	's`'	'o'	INCORRECT

Note that both 'o' characters are considered to have status WRONG_POSITION. *This is the case that's different from the official Wordle!*. In the official Wordle, only the first 'o' would be considered to be in the wrong position, while the second would be considered to be INCORRECT. However, that behaviour is a bit harder to implement, and so we have chosen a simplified definition of "WRONG_POSITION" for this assignment.

Part (a): Understanding the problem domain

Now, open the starter file `a2_part3.py`. Your first task is to implement the functions under "Part 3(a)", which will get you comfortable with the above definitions, and practice translating mathematical statements into Python code. By the end of part (a), you'll be able to take a guess (or collection of guesses) and determine the status of each guess with respect to a given answer.

The last function in this collection, `part3a_example`, gets you to visualize your work using the a small pygame visualizer that we have provided you. *This function (as well as the other "`_example`" functions) is graded, so you **must** complete it by following the instructions in the function docstring!*

Part (b): Consistency answer and automatic Wordle solvers

Computing statuses for a given guess and answer is a good start, but we can do more! In this part, we'll see how to harness the power of computation—and specifically, of *logic*, *predicates*, and *filtering comprehensions*—to implement functions that allow us to have the Pypthon interpreter "solve" a given Wordle puzzle:

> **Problem**: given a Wordle puzzle with a collection of past guesses and statuses, what are the possible answers that the puzzle could have?

As you might expect, to answer this question we'll first introduce two new definitions.

- Let W be a Wordle word set, $g \in W$ be a guess word and s be a guess status (i.e., list of elements from {CORRECT, WRONG_POSITION, INCORRECT}) with the same length as g. We say that a word $w \in W$ is a **correct answer for** g **and** s when s equals the guess status of g with respect to w.

  For example, if $g =$ 'beach', and $s = [INCORRECT, CORRECT, CORRECT, CORRECT, CORRECT]$, then:

  - The word 'teach' is a correct answer for g and s.
  - The word 'mouse' is not a correct answer for g and s.
  - The word 'reach' is another correct answer for g and s (illustrating that a given guess and status can have more than one correct answer).

- Let W be a Wordle word set, $guesses$ be a list of guess words from W, and $statuses$ a list of statuses with the same length as $guesses$ (and where each status has the same length as the guesses). We say that a word $w \in W$ is a **correct answer for** $guesses$ **and** $statuses$ when for *all* corresponding pairs of guess g_i and status s_i, w is a correct answer for g_i and s_i.

  For example, suppose $guesses =$ ['beach', 'round'], and

  $$\begin{aligned} statuses = [&[INCORRECT, CORRECT, CORRECT, CORRECT, CORRECT], \\ &[CORRECT, INCORRECT, INCORRECT, INCORRECT, INCORRECT]]. \end{aligned}$$

  Then in this case:

  - 'teach' is *not* a correct answer for these guesses and statuses, because it's not correct for the pair 'round', $[CORRECT, INCORRECT, INCORRECT, INCORRECT, INCORRECT]$).
  - 'reach' *is* a correct answer for these guesses and statuses, since it is a correct answer for both pairs of guess and status.

With these definitions in mind, complete the following questions.

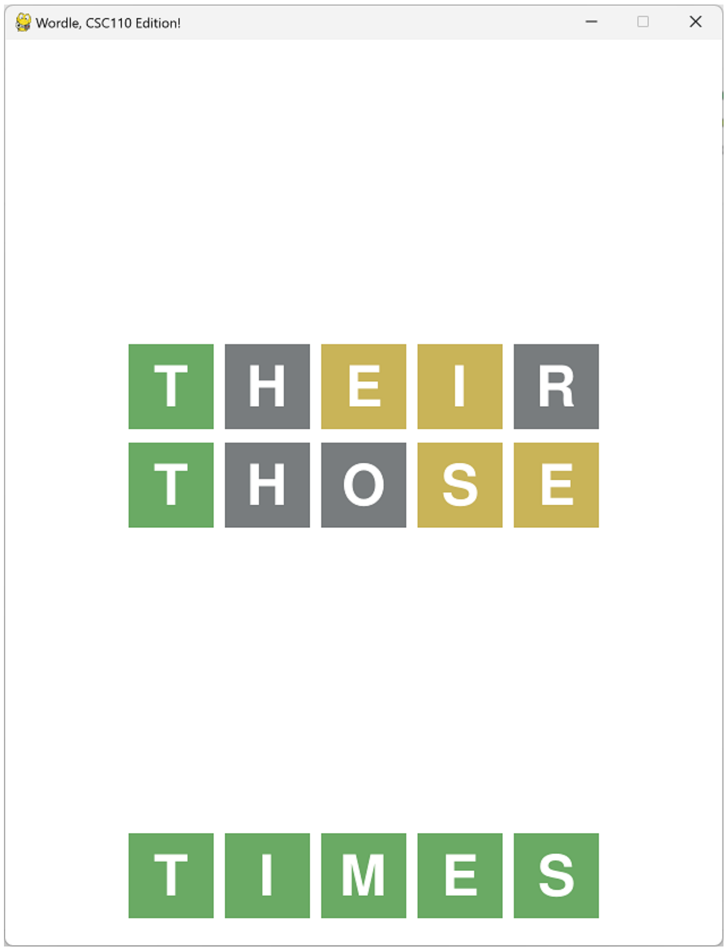
1. To make sure you understand these two definitions, implement the functions `is_correct_single` and `is_correct_multiple`, which are *translations* of these definitions into Python code.

2. Use what you've done to implement the function `find_correct_answers`, which solves the problem we described at the start of this section!

3. Visualize your work by completing the `part3b_example` function. For the first time, you'll access *real word data* contained in your `assets/word-sets` folder, which has several different word sets to pick from. Here is an example of calling this function in the Python console:

```
>>> example_guesses = ['their', 'those']
>>> example_statuses = [[CORRECT, INCORRECT, WRONG_POSITION, WRONG_POSITION,
        INCORRECT],
...                     [CORRECT, INCORRECT, INCORRECT, WRONG_POSITION,
        WRONG_POSITION]]
>>> part3b_example('assets/word-sets/possible_words_100.txt',
        example_guesses, example_statuses)
```

This should display a Pygame window like the following:



Note that using the small `possible_words_100.txt` word set, there's only one possible correct answer (`'times'`); but if you try calling this function on the larger `possible_words.txt` list, you should get more than one correct answer!

Part (c): Reverse engineering guesses

One of the key features of Wordle that made it a viral hit was the ability to share the statuses of your guesses without showing the actual guess words. For a period of time in early 2022, it seemed like Twitter was full of messages that looked like:

Wordle 455 5/6

⬜⬜⬜🟨🟨
⬜🟨🟨⬜🟨
🟨⬜🟩🟨🟩
⬜🟨🟩🟩🟩
🟩🟩🟩🟩🟩

After having solved the day's puzzle yourself, you might have looked at these statuses and thought, *"What in the world was my friend guessing?!"* It turns out that we can write a Python function to answer this question for us, without a lot of additional work!

When you defined the `is_correct_single` and `is_correct_multiple` functions earlier, you were probably thinking of the `word` parameter as the "input" to check, and the `guess` and `status` as "fixed". However, we can change our point of view and treat the `guess` parameter as the one to check, and the `word` as fixed instead.

1. Complete the function `find_correct_guesses_single`, which will let you *reverse engineer* the possible guesses that could have been made to get a given status.

2. The next function you'll complete is `find_correct_guesses_multiple`, which will let you reverse engineer the possible *lists of guesses* that could have been made to get a corresponding list of statuses. This function is a bit trickier to complete because you'll first need to compute a collection of guesses for each status individually, and then return a list containing *all possible combinations of guesses*.

Example (Reverse-engineering guesses)

Suppose we have the answer word `'later'`, and three statuses:

1. ⬜⬜⬜⬜⬜ (all INCORRECT)
2. 🟨⬜⬜🟩🟩 (WRONG_POSITION, INCORRECT, INCORRECT, CORRECT, CORRECT)
3. 🟩⬜⬜🟩🟩 (CORRECT, INCORRECT, INCORRECT, CORRECT, CORRECT)

First, we can find the possible guesses that are consistent with the answer word `'later'` and each of the three statuses individually. Suppose we do so, and obtain the following three lists of guesses (pretend that our word set is very small, just for this example!):

1. `['noisy', 'pound']`
2. `['tiger', 'tower']`
3. `['liner', 'lower']`

How can we use these lists to obtain the sequences of guesses that would give the above three statuses (and in the right order)? The key idea is that we want **all possible combinations of one guess from each of the three lists**:

```
[['noisy', 'tiger', 'liner'],
 ['noisy', 'tiger', 'lower'],
 ['noisy', 'tower', 'liner'],
 ['noisy', 'tower', 'lower'],
 ['pound', 'tiger', 'liner'],
 ['pound', 'tiger', 'lower'],
 ['pound', 'tower', 'liner'],
 ['pound', 'tower', 'lower']]
```

This is a generalization of the *Cartesian product* operation we discussed in Section 1.7 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/01-working-with-data/07-comprehensions.html), now extended to multiple sets. Formally, if we have $n \in \mathbb{Z}^+$ and sets $S_1, S_2, \dots, S_n$, then we define the Cartesian product of these sets as:

$$S_1 \times S_2 \times \cdots \times S_n = \left\{(x_1, x_2, \dots, x_n) \mid x_1 \in S_1, x_2 \in S_2, \dots, x_n \in S_n\right\}$$

You might be wondering how we will implement this operation in Python! This is possible but requires a little more advanced Python functionality, and so we've implemeted this operation for you in a helper function called `cartesian_product_general` in `a2_wordle_helpers.py`.

Your task: using `cartesian_product_general`, implement the `find_correct_guesses_multiple` function.

3. Finally, complete `part3c_example`. This will let you browse the different possible guesses you've found in a similar way as `part3b_example`. This is a great time to show off your work! 💁‍♂️

Part (d): Evaluating possible answers

Back in part (b), you were able to write a function that took lists of guesses and statuses and return all possible correct answers. However, you might be wondering—how would we actually use this function to pick our next guess in a real Wordle game?[2] Would we just pick a random possible correct answer, or is there something smarter we could do?

Intuitively, we would want to pick an answer as our next guess that, even if incorrect, has the highest chance of *revealing information* about the actual correct answer. But what do we mean by "revealing information", exactly?

Definition. Let $g \in W$ be a guess word, $a \in W$ be an answer word, and s be the guess status of g with respect to a. We define the **information score** of g with respect to a to equal:

$$info\_score(a, g) = (\#\text{ CORRECT characters in } s) + 0.5 \times (\#\text{ WRONG\_POSITION characters in } s)$$

For example, if $g =$ 'tiger' and $a =$ 'later', then the status is $s = [WRONG\_POSITION, INCORRECT, INCORRECT, CORRECT, CORRECT]$, and the information score of g with respect to a equals

$$info\_score(a, g) = 2 + 0.5 \times 1 = 2.5$$

1. In `a2_part3.py`, implement the function `information_score`, based on the above definition.

   Hint: you will find the `list.count` method useful here. (To find out more about this method, call `help` on it, or read its entry in Appendix A.2 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/A-python-builtins/02-types.html).)

2. Now, the big part! Your **final task** for this assignment is to implement the function `find_correct_answers_and_scores`. This function asks you to first find the possible correct answers for the given guesses and statuses (like in `find_correct_answers` from part 3b), but then for each answer word, treat it like a guess word and calculate its **average information score across all possible correct answers, including itself**. That might sound a bit confusing, so here's an example!

Example (average information score)

Suppose our set of possible answers is `{'reach', 'tiger', 'tower'}`. (It doesn't matter what the guesses/statuses were for this calculation.)

Then the *average information score* for `'reach'` is:

$$\frac{info\_score(\text{'reach'}, \text{'reach'}) + info\_score(\text{'tiger'}, \text{'reach'}) + info\_score(\text{'tower'}, \text{'reach'})}{3}$$
$$= \frac{5 + 1 + 1}{3}$$
$$= 2.33333\ldots$$

Similarly, the average information score for `'tiger'` is

$$\frac{info\_score(\text{'reach'}, \text{'tiger'}) + info\_score(\text{'tiger'}, \text{'tiger'}) + info\_score(\text{'tower'}, \text{'tiger'})}{3}$$
$$= \frac{1 + 5 + 3}{3}$$
$$= 3.0\ldots$$

and the average score for `'tower'` is:

$$\frac{info\_score(\text{‘reach’}, \text{‘tower’}) + info\_score(\text{‘tiger’}, \text{‘tower’}) + info\_score(\text{‘tower’}, \text{‘tower’})}{3}$$
$$= \frac{1+3+5}{3}$$
$$= 3.0\dots$$

So if these were the possible correct answers, `find_correct_answers_and_scores` would return the following dictionary (ignore possible rounding issues in this example):

```
{'reach': 2.3333333333333333, 'tiger': 3.0, 'tower': 3.0}
```

This is the most complex function on this assignment, and we **strongly recommend** breaking this down into multiple steps, and defining at least one new helper function to accomplish an intermediate step. (E.g., calculating an average information score.)

Extending your work

You've reached the end of the assignment ✨ , but you might be wanting a bit more from that last part. Even after computing these "average information scores", the possibility of creating a simple Wordle AI feels tantalizingly within reach: starting with an empty puzzle (no guesses or statuses), we can imagine an algorithm that:

1. Calls our `find_correct_answers_and_scores` function to get all possible answers and their average information scores.
2. Picks the *possible answer with the highest average information score* as the next guess.
3. Repeats steps (1) and (2) until it has found the correct answer (or run out of guesses).

This overall approach does work, and indeed is the basis for many "Wordle AIs" (or more generally, game-playing AIs) in use today. The parts that we don't yet know, and so have not included on this assignment, are: (1) how to write Python code to implement the "repeating" behaviour in Step 3—though we'll cover this in lecture soon!—and (2) whether our information score formula is the "best" one, or how to even judge whether one formula might be better than another.

If you're interested in pursuing this further please feel free to work on this on your own time after submitting the assignment, and of course we're happy to discuss this further in office hours! But for now, please move on to *submitting your work*.

Submission instructions

Please **proofread** and **test** your work carefully before your final submission! As we explain in Running and Testing Your Code Before Submission (https://q.utoronto.ca/courses/278340/pages/running-and-testing-your-code-before-submission?module_item_id=3953751), it is essential that your submitted code not contain syntax errors. **Python files that contain syntax errors will receive a grade of 0 on all automated testing components (though they may receive partial or full credit on any TA grading for assignments).** You have lots of time to work on this assignment and check your work (and right-click -> "Run in Python Console"), so please make sure to do this regularly and fix syntax errors right away.

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09/courses/1).
2. Go to Assignment 2, then the "Submissions" tab.
3. Submit the following files: `a2.tex`, `a2.pdf` (which must be generated from your `a2.tex` file), `a2_part1_q1_q2.py`, `a2_part1_q3.py`, and `a2_part3.py`. Please note that MarkUs is picky with filenames, and so your filenames must match these exactly, including using lowercase letters.
    - Please check out the Overleaf tutorial on exporting project files (https://www.overleaf.com/learn/how-to/Exporting_your_work_from_Overleaf) to learn how to download your `a2.tex` and `a2.pdf` files from Overleaf.
4. Refresh the page, and then *download each file* to make sure you submitted the right version.

Remember, you can submit your files multiple times before the due date. So you can aim to submit your work early, and if you find an error or a place to improve before the due date, you can still make your changes and resubmit your work.

After you've submitted your work, please give yourself a well-deserved pat on the back and go take a rest or do something fun or look at some art or eat some chocolate!

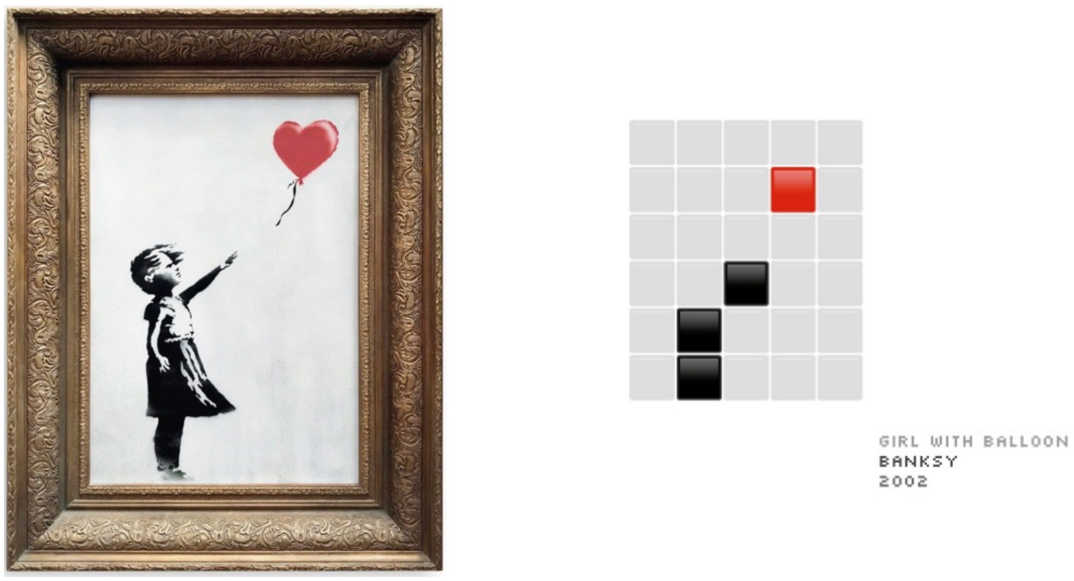


Photo credit [5x6 Art (https://twitter.com/5x6art)]

---

1. In the official Wordle game, there are actually two word sets used: a word set for all possible answers, and then a larger word set for all possible guesses. This ensures that the Wordle answers are more common English words. To keep things simple, for this assignment we will use the same word set for both answers and guesses.↩

2. Or, if you like, how would an AI bot for playing Wordle use this function?↩