EX02 - Chardle

In this assignment, we will take inspiration from the free, viral word puzzle game Wordle. If you haven't played it, you should! We do not yet have all the conceptual tools we will need to recreate this game (yet!) so this first exercise is exploring some of the fundamentals that will help ultimately make it possible.

Have you tried solving today's Wordle? If not, do so first!

In this exercise, you will prompt the user for a five-character word and a single letter. You will then test to see which indices of the word match the single letter.

You should follow the steps below for implementing the program one step at a time. To get a sense of where you are going, consider what we expect the output to be given some example inputs:

```
$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: h
Searching for h in hello
h found at index 0
1 instance of h found in hello

$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: l
Searching for l in hello
l found at index 2
l found at index 3
2 instances of l found in hello

$ python -m exercises.ex02_chardle
Enter a 5-character word: weeee
Enter a single character: e
Searching for e in weeee
e found at index 1
e found at index 2
e found at index 3
e found at index 4
4 instances of e found in weeee

$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: z
Searching for z in hello
No instances of z found in hello

$ python -m exercises.ex02_chardle
Enter a 5-character word: tarheels
Error: Word must contain 5 characters

$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: hello
Error: Character must be a single character.
```

Part 0. Setting up the Python Program

In Visual Studio Code, be sure your workspace is open. (Reminder: File > Open Recent > comp110-YYS-workspace is a quick way to reopen it! Where YY is the current year and S is the semeseter: S for Spring, F for Fall.)

Open the Explorer pane (click the icon with two sheets of paper or to to *View > Explorer*) and expand the *Workspace* directory.

Overview

- `ex02_chardle.py`

Before you begin working on the program, you should add a *docstring* to the top of your Python *module* just as you did in past exercises. Then, you should add a line with the special variable named `__author__` assigned to be a string with your 9-digit student PID. (Disclaimer: Out in the real world the `__author__` variable is typically your name and e-mail address, but since we will grade your programs we'd like to avoid potential bias in seeing your names as part of the programs as we're grading.) Add the following lines at the top of your file. Fill in *your* 9-digit UNC PID number, without any spaces or dashes, in the `__author__` string as seen below.

```
"""EX02 - Chardle - A cute step toward Wordle."""

__author__ = "1234567890"
```

Important: As you progress through the exercise, comment your code! As a general rule, if 2 or more minutes are spent thinking about how to write a particular line or block of code, it's a good idea to add a comment. Commenting is mandatory as it should be an integral part of the problem-solving process. More information on how commenting will be graded can be found in part 7.

Part 1. Prompting for an Input Word

The first task of this program is to create a function, `input_word`, that will gather an input from the user. It has *no* parameters and a `str` return type! Instead of a parameter, it'll just use user input to determine a local variable (which can only be accessed inside of the function's body).

Your function is responsible for asking the user to enter a 5-character word. This input should be stored as a `str` local variable within your `input_word` function. **Please choose meaningful, descriptive names for your variables.**

Write your prompts and diagnostic message such that you can reproduce the following in the shell after saving and running your program. Here is an example usage:

```
>>> input_word()
Enter a 5-character word: camp
Error: Word must contain 5 characters.
'camp'

>>> input_word()
Enter a 5-character word: horsey
Error: Word must contain 5 characters.
'horsey'

>>> input_word()
Enter a 5-character word: ducky
'ducky'
```

If the input word has a length of 5, it should return that input word. If the input word has a length of anything other than 5, it should return that input word, but first it should print: `Error: Word must contain 5 characters.`

As seen in the final example usage (where the user entered 'ducky'), the word *length* was valid, so the function `return`ed the word the user entered without printing the error message.

So far, we have not implemented the 'checking' system for when a word is of valid length. To do this, once you have gathered the input, the function should also check *if* the length of the word is exactly 5 characters. You will need to use the `len` function, as described in the Objects and Data Types lesson, in order to find the length of your input strings. Think about how you should implement this. Should you check that every word entered is of length 5 characters? Why or why not? Think about the *if* of each scenario. *If* the user input is 5 characters long, what would you like to do? *If* the user input is not 5 characters long, what would you like to do?

WARNING: Autograding will very specifically be looking for *exactly* the format of lines output shown above. You will not see the `$` at your command-line prompt in VSCode, you can ignore that part.

Part 2. Prompting for an Input Character

Now that we can obtain a word from the user and verify its length, we will do the same for obtaining a character. Create a function called `input_letter` that gathers input from the user. Like the `input_word` function, `input_letter` should prompt the user to enter a single character. This input should be stored as a local `str` variable within your `input_letter` function. Below is an example usage you should be able to reproduce exactly:

```
>>> input_letter()
Enter a single character: eeee
Error: Character must be a single character.
'eeee'

>>> input_letter()
Enter a single character: rawr
Error: Character must be a single character.
'rawr'

>>> input_letter()
Enter a single character: r
'r'
```

Part 3. Checking Indices for Matches

Now that we are able to gather input from `input_word` and `input_letter`, our next task is to check if the input character matches any of the characters within the input word. Begin by defining a function named `contains_char`. This function will take two parameters: one that will store the input word recieved from `input_word` and another for the letter guess recieved from `input_letter`, both of which are strings. The function won't return any value. You will then need to check each index of the word to see if it matches (hint: *is equal to*) the character.

Soon you will learn a more efficient way of performing these checks using a concept called a *loop*. For now, you will need to check each of the five indices of the word string to see if it is equal to the character string. *If* so, *then* you should print out a message indicating the letter being searched for was found at a given index. Your goal in this part is to be able to do the following:

```
>>> contains_char(word="kitty", letter="z")
Searching for z in kitty

>>> contains_char(word=input_word(), letter=input_letter())
Enter a 5-character word: ponds
Enter a single character: ducks
Error: Character must be a single character.
Searching for ducks in ponds

>>> contains_char(word=input_word(), letter=input_letter())
Enter a 5-character word: ponds
Enter a single character: ducks
Error: Character must be a single character.
Searching for ducks in ponds

>>> contains_char(word=input_word(), letter=input_letter())
Enter a 5-character word: pzazz
Enter a single character: z
Searching for z in pzazz
z found at index 1
z found at index 3
z found at index 4
```

Part 4. Counting Matching Indices

matching characters. Before you find any matching characters, think about what value this variable should start with. Initialize this variable to that starting value. Each time you find a match, increase your counting variable's value by 1. Finally, print out a message that indicates how many instances of the character you found in the input word.

You will need to exactly match the output formatting of the examples below. Specifically, notice that there are different messages depending on whether no matches are found, a singular match is found, or multiple matches are found (i.e., *No instances* vs. *1 instance* vs. *2 instances*).

```
>>> contains_char("kitty", "z")
Searching for z in kitty
No instances of z found in kitty

>>> contains_char(word=input_word(), letter=input_letter())
Enter a 5-character word: ponds
Enter a single character: ducks
Error: Character must be a single character.
Searching for ducks in ponds
No instances of ducks found in ponds

>>> contains_char(word=input_word(), letter=input_letter())
Enter a 5-character word: ponds
Enter a single character: ducks
Error: Character must be a single character.
Searching for ducks in ponds
No instances of ducks found in ponds

>>> contains_char(word=input_word(), letter=input_letter())
Enter a 5-character word: pzazz
Enter a single character: z
Searching for z in pzazz
z found at index 1
z found at index 3
z found at index 4
3 instances of z found in pzazz
```

Part 5. Exiting Early for Invalid Inputs

What happens if you input a word with fewer than 5 characters? Or you input a "char" that is more than one character, or none at all? As seen in part 3, when the user put in 'ducks' as the single character, they were able to recieve the error message but the code continued to run as the call to `contains_char` was still executed. We don't want this to happen. It's good practice to handle bad input from an end-user gracefully in your programs. Our strategy, for now, will be to modify both `input_word` and `input_letter` to exit the program early after it prints the error message but *before* it returns something.

There is a special built-in function called `exit()` that will send a signal to your system and tell the program to quit at that point, not continuing on further in the program. Before attempting to implement the following behavior in your program, think about where it logically *makes sense* to try adding these checks for correct input.

Here is how your program should work after completing this part:

```
>>> input_word()
Enter a 5-character word: pzazzzz
Error: Word must contain 5 characters.

>>> input_letter()
Enter a single character: tar
Error: Character must be a single character.
```

Part 6. Putting everything together

example usages), the `main` function simplifies this process. When called, it automatically handles all the function calls and variable assignments needed to run the game smoothly.

Start by defining a function named `main` that does not take any parameters as input and does not return any value. It should simply call the functions we just defined:

```
contains_char(word=input_word(), letter=input_letter())
```

Here is how your program should work after completing this part:

```
>>> main()
Enter a 5-character word: lives
Enter a single character: l
Searching for l in lives
l found at index 0
1 instance of l found in lives

>>> main()
Enter a 5-character word: elephant
Error: Word must contain 5 characters.
```

Once you have your `main` function and game loop working, there's only one bit of icing left to add to your delicious code cake. We will fully explain what is going on in the following code snippet soon, but for now note that this is an idiom common to Python programs like the one you have written. We will learn it does two things: 1. it makes it possible to run your Python program as a module (if you tried `python -m exercises.ex02_chardle` right now you would see nothing happens), and 2. it makes it possible for other modules to *import* your functions and reuse them. Add the following snippet of code as the last 2 lines of your program (notice, there are two underscores *on both sides* of the words `name` and `main`):

```
if __name__ == "__main__":
    main()
```

Now you can try running your game as a module and it should work: `python -m exercises.ex02_chardle`.

```
$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: h
Searching for h in hello
h found at index 0
1 instance of h found in hello

$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: l
Searching for l in hello
l found at index 2
l found at index 3
2 instances of l found in hello

$ python -m exercises.ex02_chardle
Enter a 5-character word: weeee
Enter a single character: e
Searching for e in weeee
e found at index 1
e found at index 2
e found at index 3
e found at index 4
4 instances of e found in weeee

$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: z
```

```
$ python -m exercises.ex02_chardle
Enter a 5-character word: tarheels
Error: Word must contain 5 characters.

$ python -m exercises.ex02_chardle
Enter a 5-character word: hello
Enter a single character: hello
Error: Character must be a single character.
```

Part 7. Style and Documentation Requirements (Manually Graded)

We will manually grade your code and are looking for good choices of meaningful variable names. Your variable names should be descriptive of their purposes. You should also use the Python `snake_case` convention where variable names are all lowercase and new words are separated by underscores. You should not make excessive use of variables where single variables will suffice (e.g. you only need one counting variable for counting the instances in Part 3).

As part of the manual grading, we will be looking to see if you have commented on your code! No comments imply that there were no challenges or moments spent considering how to approach a code. It is quite rare to complete every exercise, challenge question, and practice problem without engaging in some form of problem-solving. Even the most experienced programmers use a piece of paper to map out their approach when working on practice problems, often leading to comments being added to code.

As a general rule, if 2 or more minutes are spent thinking about how to write a particular line or block of code, it's a good idea to add a comment. Explain what's happening on that line, how the solution was reached, the reasoning behind the approach, or provide a note for future reference to recall the problem-solving steps. If you received help from office hours or tutoring, go back to the code you were stuck on and explain it to yourself. If you see that you need a second to understand what is going on, comment!

Comments don't need to be extensive, but they should reflect a genuine effort to explain the process in your own words. Commenting should be an integral part of the problem-solving process.

Permitted Constructs

We expect you to implement this exercise using only the concepts covered in COMP110. If you have prior programming experience, restrict your implementation to only the concepts covered. While there are many ways to implement this program with additional concepts beyond those we have covered, you should not attempt to do so until after submitting this exercise for full credit once the autograder is posted. Gaining additional practice with the fundamentals may feel clunky, but will help ensure you have full command over the concepts we expect you to know. Additionally, it is good practice for working in other programming environments which are more constrained and require creativity to overcome restrictions.

Part 8. Type Safety and Linting

The autograder uses industry standard tools for checking the type safety of your programs (e.g. being sure you're using variables of the correct data types in valid ways) and linting style rules. If you have point deductions on Type Safety or Linting, read through the feedback the autograder gives and it should direct you to the line number in your program with the issue.

Make a Backup Checkpoint "Commit"

As you make progress on this exercise, making backups is encouraged.

1. Open the Source Control panel (Command Palette: "Show SCM" or click the icon with three circles and lines on the activity panel).

that plus symbol to add all changes to the next backup. You will now see the files listed under "Staged Changes".
    - If you do not want to backup *all* changed files, you can select them individually. For this course you're encouraged to back everything up.
4. In the Message box, give a brief description of what you've changed and are backing up. This will help you find a specific backup (called a "commit") if needed. In this case a message such as, "Progress on Exercise 1" will suffice.
5. Press the Check icon to make a *Commit* (a version) of your work.
6. Finally, press the Ellipses icon (...), look for "Pull/Push" submenu, and select "Push to...", and in the dropdown select your backup repository.

Submit to Gradescope for Grading

All that's left now is to hand-in your work on Gradescope for grading!

Login to Gradescope and select the assignment named "EX02 - Chardle". You'll see an area to upload a zip file. To produce a zip file for autograding, return back to Visual Studio Code.

If you *do not* see a Terminal at the bottom of your screen, open the Command Palette and search for "View: Toggle Integrated Terminal".

Type the following command (all on a single line):

```
python -m tools.submission exercises/ex02_chardle.py
```

In the file explorer pane, look to find the zip file named "yy.mm.dd-hh.mm-exercises-ex02_chardle.py.zip". The "yy", "mm", "dd", and so on, are timestamps with the current year, month, day, hour, minute. If you right click on this file and select "Reveal in File Explorer" on Windows or "Reveal in Finder" on Mac, the zip file's location on your computer will open. Upload this file to Gradescope to submit your work for this exercise.

Autograding will take a few moments to complete. For this exercise there will be 10 "human graded" points. Thus, you should expect a maximum autograding score of 90 possible points on this assignment. If there are issues reported, you are encouraged to try and resolve them and resubmit. If for any reason you aren't receiving full credit and aren't sure what to try next, come give us a visit in office hours!

Contributor(s): Kris Jordan, Viktorya Hunanyan, Alyssa Lytle, Izzi Hinks