Lesson 19. Capstone project: Scrabble, Art Edition

After reading lesson 19, you'll be able to

- Apply conditionals and loops to write a more complicated program
- Understand what's being asked of you in a program
- Draw up a plan of how to solve a problem before starting to code
- Break the problem into smaller subproblems
- Write code for the solution

You're playing a simpler version of Scrabble with your kids. The kids have been winning most games so far, and you realize it's because you aren't picking the best word from the given tiles. You decide that you need a bit of help in the form of a computer program.

The problem

Write a program that can tell you words that you can form from a set of tiles; the set of all valid words is a subset of all the English words (in this case, only words related to art). When dealing with choosing the best word from the given tiles, here are some details to remember:

- All valid words related to art are given to you as a string, each word separated by a newline. The string organizes the words by length, shortest to longest. All valid words contain only letters in the alphabet (no spaces, hyphens, or special symbols). For example,

```
"""art
hue
ink
oil
pen
wax
clay
draw
film
...
crosshatching
"""
```

- The number of tiles you get can vary; it's not a fixed number.
- Letters on tiles don't have point values; they're all worth the same.
- The tiles you get are given as a string. For example, `tiles = "hijklmnop"`.
- Report all valid words you can form with your tiles in a tuple of strings; for example, `('ink', 'oil', 'kiln')`.

19.1. UNDERSTANDING THE PROBLEM STATEMENT

This programming task sounds involved, so try to break it into a few sub-tasks. There are two big parts to this problem:

- Represent all the possible valid words in a format that you can work with. Convert the words from a long string of characters into a tuple of string words.
- Decide whether a word in the list of all valid words can be made with the set of tiles you're given.

19.1.1. Change the representation of all valid words

Let's tackle the first part, which will help you create a tuple of all the valid words so you can work with them later. You need to do this step because if

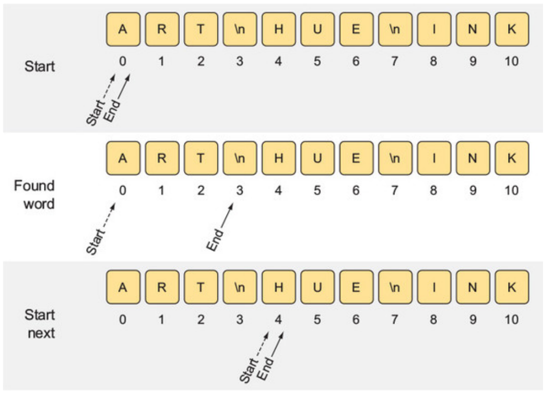
The set of all valid words is given to you as a string. To the computer, each "line" that a human sees isn't a line with a word on it, but a long sequence of characters.

Draw out the problem

It's always a good idea to start with a small sketch of what you need to do. To the human eye, the string looks nicely organized, and you can tell the words apart, but the computer doesn't know the concept of words in a string, only single characters. The computer sees something like `"""art\nhue\nink\noil\n...\ncrosshatching"""`.

The line breaks that you can see with your eyes are single characters themselves, each called the *newline* (or *linebreak*) character, represented by `\n`. You'll have to find out the position of every newline character so you can separate each word. Figure 19.1 shows how you might think about this in a more systematic way.

Figure 19.1. Converting a string of characters into words. In the top example named *start*, you can keep track of where you are in the character string by using a *start* and *end* pointer. In the middle example named *found word*, you stop changing *end* when you reach a newline character. In the bottom example named *start next*, you reset the *start* and *end* pointers to the character right after the newline.



With this simple sketch, you can already see how to achieve this task. The *start* and *end* pointers start at the beginning of the big string. As you're looking for a newline character to mark the end of the word, you'll increment the *end* pointer until you find the `\n`. At that point, you can store the word from the *start* pointer to the *end* pointer. Then, move both pointers to one index past the newline character to start looking for the next word.

Come up with some examples

Write some test cases that you may want to think about as you're writing your program. Try to think of simple cases and complex ones. For example, all valid words might be just one word, such as `words = """art"""`, or it might be a few words, such as the example given in the problem statement.

Abstract the problem into pseudocode

Now that you have an idea of how to convert characters to words, you can start writing a mixture of code and text to help you put the big picture into place, and to start thinking about the details.

Because you need to look at all letters in the string, you need a loop. In the loop, you decide whether you've found a newline character. If you found the newline character, save the word and reset your pointer indexes. If you didn't find a newline character, keep incrementing only the *end* index until you do. The pseudocode might look like this:

```
word_string = """art
hue
ink
"""
set start and end to 0
set empty tuple to store all valid words
for letter in word_string:
    if letter is a newline:
        save word from start to ends in tuple for all
        reset start and end to 1 past the newline cha
    else:
        increment end
```

19.1.2. Making a valid word with the given tiles

Now you can think about the logic for deciding whether you can make a valid word using the given tiles, with the valid word coming from the list of allowed words.

Draw out the problem

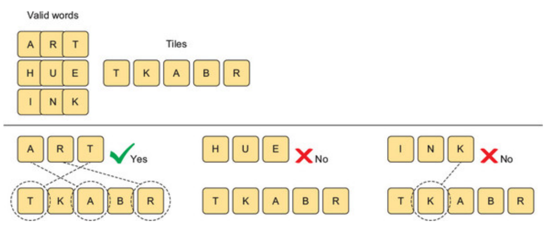
- You can start by looking at the tiles in your hand. Find all combinations of them. Then you can look at each combination of letters and see whether they match any of the valid words.
- You can start by looking at the valid words and see whether each can be made using the tiles you have.

Thinking like a programmer

This part is crucial when you're trying to decide which way to approach a problem. The process of drawing helps you think of a few ways before settling on one. If you start to code immediately, you'll feel boxed into one path that may or may not even be appropriate for the problem at hand. The process of sketching will help you see what issues may arise with a few solutions without committing to any yet.

The first option, although possibly more intuitive, is a bit harder to implement with what you know so far because it involves finding all combinations and permutations of all the tiles. The second option is more appropriate at this time. Figure 19.2 illustrates the second option.

Figure 19.2. Given the valid words and a set of tiles, start with the first valid word and check whether all its letters are in the set of tiles. If so, add it to the set of words you can make. If at least one letter is in a valid word but not in the tiles, you can't make the word.



You'll go through each valid word and then look at each letter in that word, checking whether you can find the letter in your tiles. After you've gone through all the letters of the word and you're able to find them in your tiles, you can make that word with your tiles. As soon as you find that there's one letter not in your tiles, you can immediately stop because you can't make that word.

Come up with some examples

Coming up with examples can help you determine special situations you may have to take care of in your code. Here are some examples of tiles you may want to make sure your code can handle:

- A single tile—in this case, you can't make any valid word.
- All the tiles given make exactly one valid word—with `tiles = "art"`, you can make the word *art*.
- All the tiles given make exactly two valid words—with `tiles = "euhtar"`, you can make *art* and *hue*.
- You can make one valid word but have extra tiles left over—with `tiles = "tkabr"`, you can make *art* and have *k* and *b* left over.
- You have only one tile of a certain letter, but a valid word uses two of that letter—with `tiles = "colr"`, you can't make the word *color* because you have only one *o*.

Abstract the problem into pseudocode

With pseudocode, you can start to think about more of the details that you discovered while coming up with examples. You'll need to go through each valid word to see whether you can make it with your tiles, so you'll need a loop. Then you'll go through each letter in that word; you'll need a nested loop inside the first one. You can immediately exit the inner loop as soon as you find one letter that isn't in your tiles. But if each letter you look at is in your tiles, keep going.

This logic has two tricky parts: (1) how to keep track of words that have multiples of the same letter and (2) how to tell when you found the full word in your tiles. You don't have to outline exactly how to do these in the pseudocode, but you should be able to tell whether they're issues that can be resolved. I can tell you that they can be resolved, and you'll see how in the next section. The pseudocode for this part might be

```
for word in valid_words:
    for letter in word:
        if letter not in tiles:
            stop looking at remaining letters and go
        else:
            remove letter from tiles (in case of dupl
    if all letters in valid word found in tiles:
```

Notice that there's a lot going on and a few more variables to keep track of in this problem than you're used to! Without thinking about the problem first, you'd quickly get lost. At this point, with an understanding of the major components to this problem, you can start to write the code. An important first step is deciding how to divide your code into smaller, more manageable chunks.

Thinking like a programmer

Dividing code into smaller pieces is a necessary and important skill for a programmer for a few important reasons:

- Large problems look less intimidating after they're broken into smaller pieces.
- Pieces are easier to code when you can focus on only the relevant parts of the problem.
- Pieces are much easier to debug than an entire program, because the number of possible inputs to a module is typically a lot smaller than the number of possible inputs to your entire program.

When you know that each separate piece works as expected, you can put them together to create your final program. The more you program, the more you'll get the hang of what would make a good, coherent piece of code.

19.2. DIVIDING YOUR CODE INTO PIECES

You can now start thinking about how to divide the code into small chunks of logic. The first chunk is usually to look at the input given and extract all the useful information you want to use.

- Set up the valid words related to art (as a string) and set up the tiles you're starting out with (as a string).
- Set up initializations for *start* and *end* pointers to find all valid words.
- Set up an empty tuple to add to it all valid words as you find them.
- Set up an empty tuple for the words found in your tiles.

Listing 19.1 provides the code for these initializations. You'll notice something new: a string variable that contains characters within two sets of triple quotes. The triple quotes allow you to create a string object that spans multiple lines. All characters inside the triple quotes are part of the string object, including line breaks!

Listing 19.1. Scrabble: Art Edition code for initializations

```
words = """art                          1
hue
ink
...
crosshatching
"""
tiles = "hijklmnop"

all_valid_words = ()                 2
start = 0                            3
end = 0                              4
found_words = ()                     5
```

- ***1*** **Valid words as a big string**
- ***2*** **Empty tuple for all valid words**
- ***3*** **Initializes a pointer to the beginning of index search**
- ***4*** **Initializes a pointer to the end of index search**
- ***5*** **Empty tuple for words found in tiles**

In this program, the second chunk of logic is to convert the big string of all words into a tuple containing string elements as each word. With the pseudocode written previously, all you have to do is convert the English parts to code. The one part you must be careful of is how to add the valid word to your tuple of valid words. Notice that the word you find is a single word that's added to the tuple, so you'll have to use the concatenation operator between your valid words tuple and the singleton tuple of the word that you just found.

Listing 19.2 shows the code. Pointers `start` and `end` are initially 0, pointing to the first character. Words are read as one big string, so you iterate through each character. When the character is a newline, you know that you've reached the end of a word. At that point, you save the word by indexing using the `start` and `end` pointer positions. Then, reset the pointers to be one position past the newline's position; this is the character that starts the next valid word. If the character isn't a newline, you're still reading what word it is, so move only the `end` pointer over.

```
for char in words:
    if char == "\n":
        all_valid_words = all_valid_words + (words[st

        start = end + 1
        end = end +1
    else:
        end = end + 1
```

- *1* **Iterates through every character**
- *2* **Checks whether the character is a newline**
- *3* **Adds singleton tuple to current all valid words tuple**
- *4* **Moves start and end pointers to start of next word**
- *5* **Moves only end pointer**

The third and last chunk of logic is to check whether each of the valid words can be made using your tiles. As with the previous chunk, you can copy the pseudocode and fill in the blanks. A couple of interesting things to note were left unanswered in the pseudocode: (1) how to keep track of words that have multiples of the same letter and (2) how to tell when you found the full word in your tiles.

To solve (1), you can write code that removes tiles as they're matched from a valid word. With each new valid word, you can use a variable named `tiles_left`, initially all the tiles you have, that keeps track of the tiles you have left. As you iterate through each letter in a valid word and find that it's in your tiles, you can update `tiles_left` to be all the letters except the letter just found.

To solve (2), you know that if you found all the tiles and you've been removing tiles from `tiles_left` as you find them, then the number of tiles removed plus the length of the valid word are going to be equal to the number of tiles you started with.

The following listing shows the code. There's a nested loop in this code. The outer one goes through each valid word, and the inner one goes through each letter for a given valid word. As soon as you see a letter in a word that's not in your tiles, you can stop looking at this word and go on to the next one. Otherwise, keep looking. The variable `tiles_left` stores the tiles you have left after checking whether a letter from a valid word is in your tiles. Every time you find that a letter is in your tiles, you get its position and make a new `tiles_left` with all the remaining letters. The final step is to check whether you made a full, valid word with your tiles by using up all the letters. If so, add the word.

Listing 19.3. Game code to check whether valid words can be made with tiles

```
for word in all_valid_words:
    tiles_left = tiles
    for letter in word:
        if letter not in tiles_left:
            break
        else:
            index = tiles_left.find(letter)
            tiles_left = tiles_left[:index]+tiles_lef
    if len(word) == len(tiles)-len(tiles_left):
        found_words = found_words + (word,)
print(found_words)
```

- *1* **Looks at every valid word**
- *2* **tiles_left deals with duplicate tiles**
- *3* **Looks at every letter in a valid word**
- *4* **Stops looking if letter not in tiles_left**
- *5* **Finds position of letter in tiles_left**
- *6* **Removes letter and makes a new tiles_left**
- *7* **Checks whether found entire word**
- *8* **Adds word to found_words**

At the end, you print all the words you can make. But you can tweak the results you get to choose words that are only a certain length, or only words that are the longest, or words that contain a certain letter (whatever you prefer).

SUMMARY

In this lesson, my objective was to show you how to think about approaching complicated problems and to walk you through a real-life problem for which you could use programming to write a custom program for your situation. Understanding a problem before coding can be a major confidence boost. You can use pictures or simple input values and expected outputs to help refine your understanding of the problem.

When you understand the problem, you should write a few pieces of pseudocode. You can use a mixture of English and code to see whether you'll have to break up the problem further before starting to code.

The final step is to look at the visual representation and the abstractions you came up with and use these as natural divisions in your code. These smaller pieces are easier to handle when coding, and they also provide natural points for taking a break from coding to test and debug the code.

Here are the key takeaways:

- Understand the problem being asked by drawing a couple of relevant pictures.
- Understand the problem being asked by coming up with a few simple test cases that you can write out.
- Generalize parts of the problem to come up with formulas or the logic for accomplishing each part.
- Pseudocode can be useful, especially for writing algorithm logic that includes conditionals or looping constructs.
- Think in terms of code pieces and ask whether the code has any natural divisions—for example, initializing variables, implementing one or more algorithms, and cleanup code.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 18. Repeating tasks while conditions hold

NEXT
Unit 5. Organizing your code into reusable blocks