🤖

Assignment 3 - Scrabble Solver

The “right” move.

The goal of this app is to develop an `agent` to find the optimal **initial** move of a Scrabble game, given the following inputs:

- An empty Scrabble board.
    - Assume the agent knows how the premium squares work & locations.
- The complete SOWPODS word list, containing all 267,751 legal words.
- A collection of 7 tiles in the agent’s rack.
    - At most 2 may be wildcard tiles.
    - `rack` is represented by one line of a flat text file, uppcase letters used to represent A-Z and an `_` underscore character to represent a `wildcard`

We should be outputting the `rack` (starting 7), the `board` the `points` total, and the `elapsed_time` it took to get the answer back to the user.

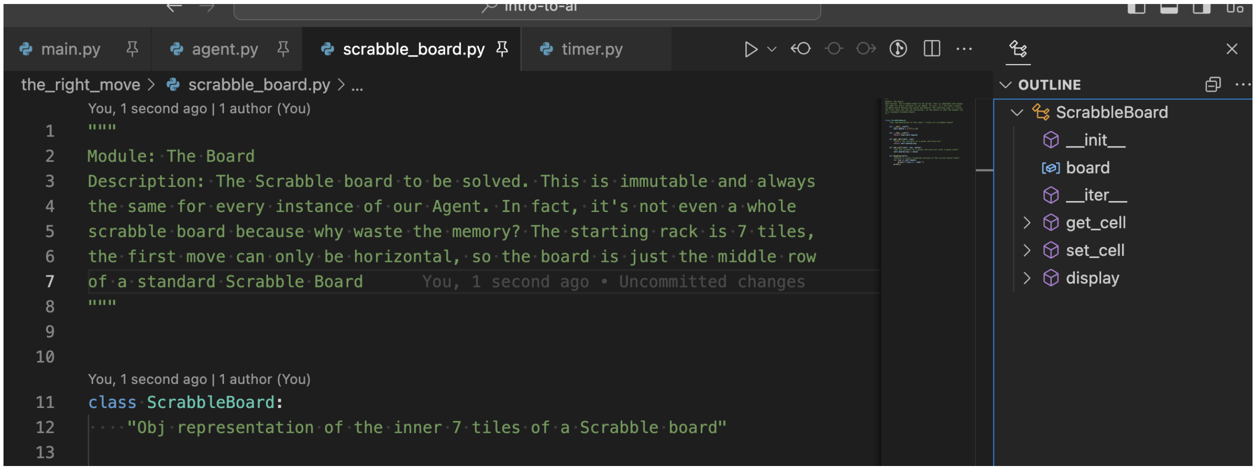
We should strive to

- minimize complicated computations (ie, use heuristics to reduce our data size)
- memory requirements - this is the most complicated one yet
- time - thats the biggest measure of our efficiency here

The main approach I want to take here is basically the same as I have taken for the other projects: build the bones of the app, solve it in the simplest way possible, get more efficient from there.

When building out the bones of the app we can take into account some of the information in the problem prompt in order to narrow our agents scope as much as possible.

The Board



```
"""
Module: The Board
Description: The Scrabble board to be solved. This is immutable and always
the same for every instance of our Agent. In fact, it's not even a whole
scrabble board because why waste the memory? The starting rack is 7 tiles,
the first move can only be horizontal, so the board is just the middle row
of a standard Scrabble Board
"""



class ScrabbleBoard:
    "Obj representation of the inner 7 tiles of a Scrabble board"
```

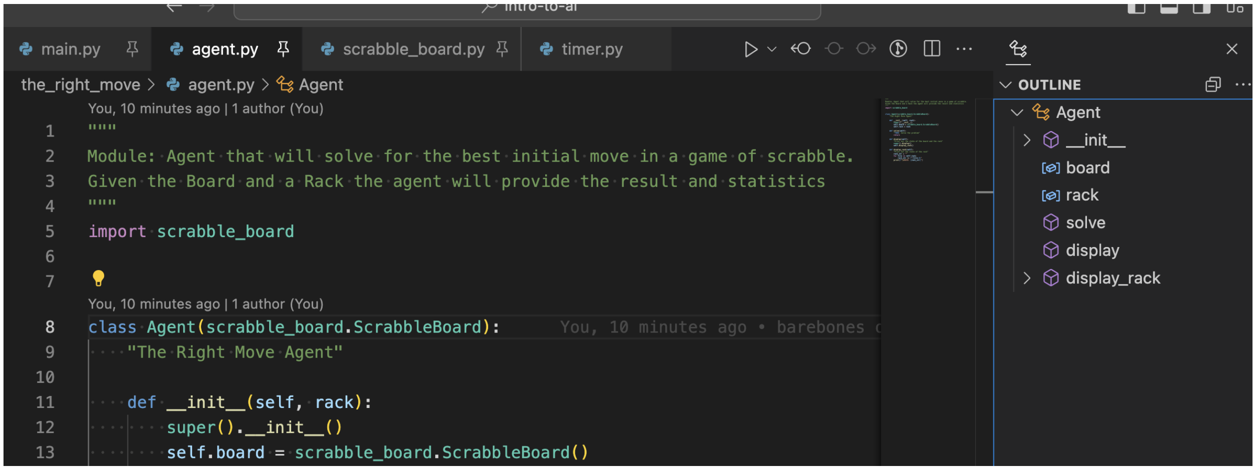
So right off the bat we can reduce the scope our agent searches by eliminating the rest of the board. The first move can only be horizontal, so thats 13 cells of a simple list.

I want to represent the empty cell with `"X"`

There is a simple `setter` method for setting a value in a cell, the `init` method instantiates a board - for now with no premium cells. A display method for printing the state of the board (in a pleasant way!)

The Agent

So again for now lets start with the barebones. An agent that `extends` the Board (because the board is useless on its own), instantiates a `rack` and displays it (pleasantly). Here is what we've got:


intro-to-ai
main.py
agent.py
scrabble_board.py
timer.py
the_right_move > agent.py > Agent
You, 10 minutes ago | 1 author (You)
You, 10 minutes ago | 1 author (You)
You, 10 minutes ago • barebones c
OUTLINE
Agent
__init__
board
rack
solve
display
display_rack

```python
"""
Module: Agent that will solve for the best initial move in a game of scrabble.
Given the Board and a Rack the agent will provide the result and statistics
"""
import scrabble_board


class Agent(scrabble_board.ScrabbleBoard):
    "The Right Move Agent"

    def __init__(self, rack):
        super().__init__()
        self.board = scrabble_board.ScrabbleBoard()
```

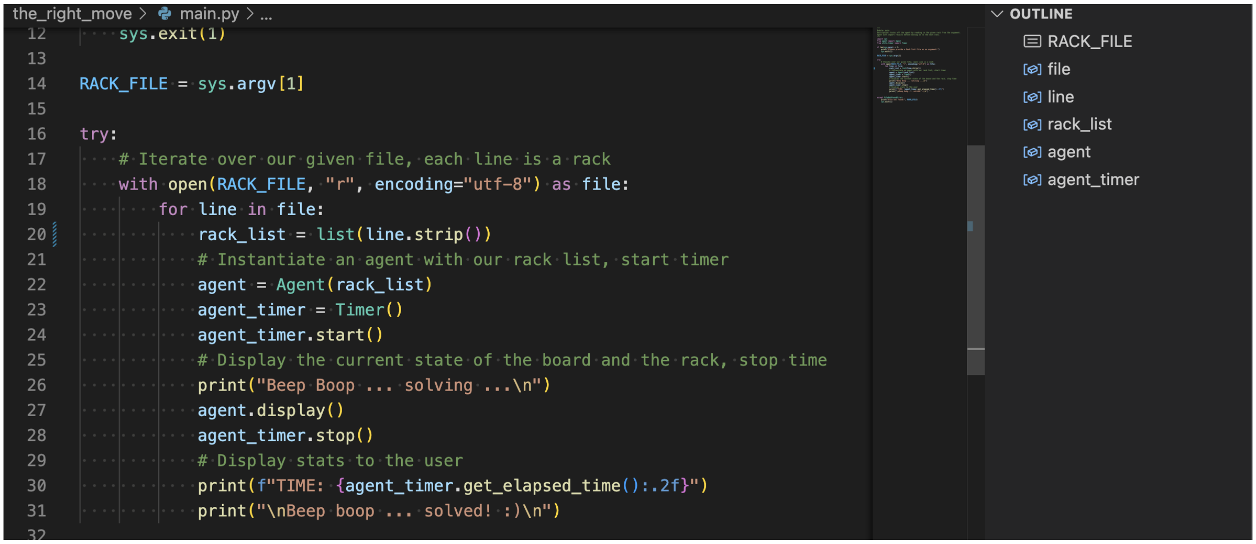
Main

Lets test it out!

Logically it makes sense to start with reading in the `rack`. As described it will be a text file where each line is a group (note: not set!) of 7 `tiles` : `A-Z` and `_` for `wildcard`

- The agent will do its thing, then report results.
- It will then move on to the next `rack` until the file has been solved.

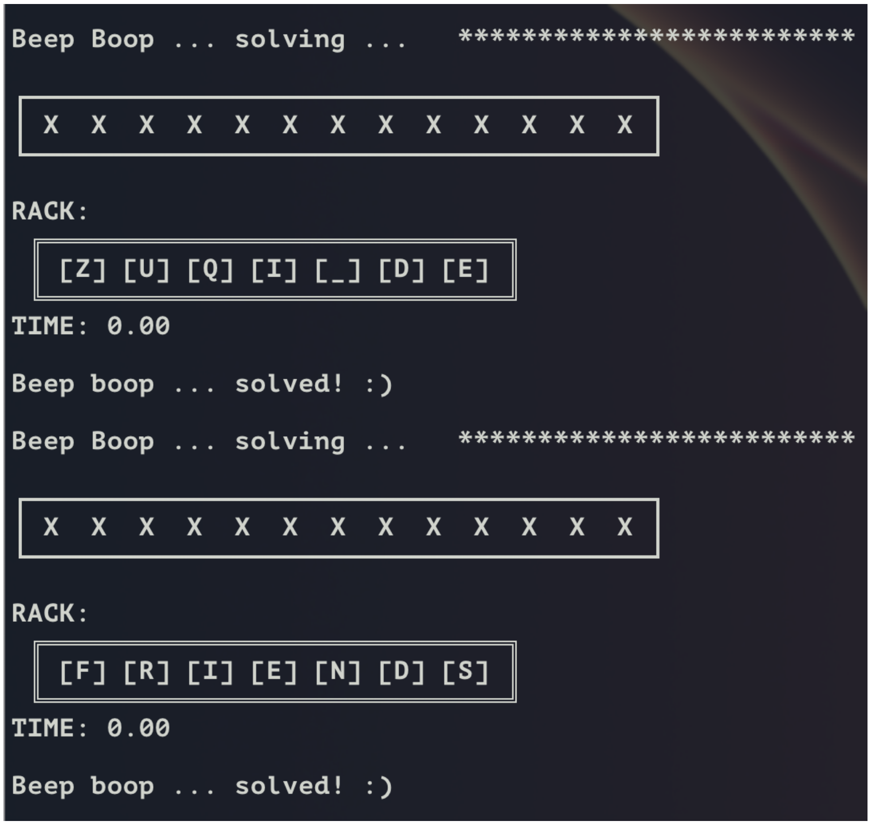
We can also steal our `Timer` class from the last assignment.



```python
    sys.exit(1)

RACK_FILE = sys.argv[1]

try:
    # Iterate over our given file, each line is a rack
    with open(RACK_FILE, "r", encoding="utf-8") as file:
        for line in file:
            rack_list = list(line.strip())
            # Instantiate an agent with our rack list, start timer
            agent = Agent(rack_list)
            agent_timer = Timer()
            agent_timer.start()
            # Display the current state of the board and the rack, stop time
            print("Beep Boop ... solving ...\n")
            agent.display()
            agent_timer.stop()
            # Display stats to the user
            print(f"TIME: {agent_timer.get_elapsed_time():.2f}")
            print("\nBeep boop ... solved! :)\n")
```

So, we read in the file as an argument, instantiate our `agent` with it's `rack` and start the timer. Right now all we can do is display the `board` and `rack`

Here's what it looks like



```
Beep Boop ... solving ...   *************************

┌────────────────────────────────────────┐
│ X  X  X  X  X  X  X  X  X  X  X  X  X  │
└────────────────────────────────────────┘

RACK:
 ╔═════════════════════════════╗
 ║ [Z] [U] [Q] [I] [_] [D] [E] ║
 ╚═════════════════════════════╝
TIME: 0.00

Beep boop ... solved! :)

Beep Boop ... solving ...   *************************

┌────────────────────────────────────────┐
│ X  X  X  X  X  X  X  X  X  X  X  X  X  │
└────────────────────────────────────────┘

RACK:
 ╔═════════════════════════════╗
 ║ [F] [R] [I] [E] [N] [D] [S] ║
 ╚═════════════════════════════╝
TIME: 0.00

Beep boop ... solved! :)
```

Okay so lets fill out a little bit more functionality. We can add in most everything that is required to physically (digitally) solve the first move. We will need the following:

- The official scrabble dictionary
- An anagram creator
- A calculator

The anagram creator will likely be the real meat of the program.

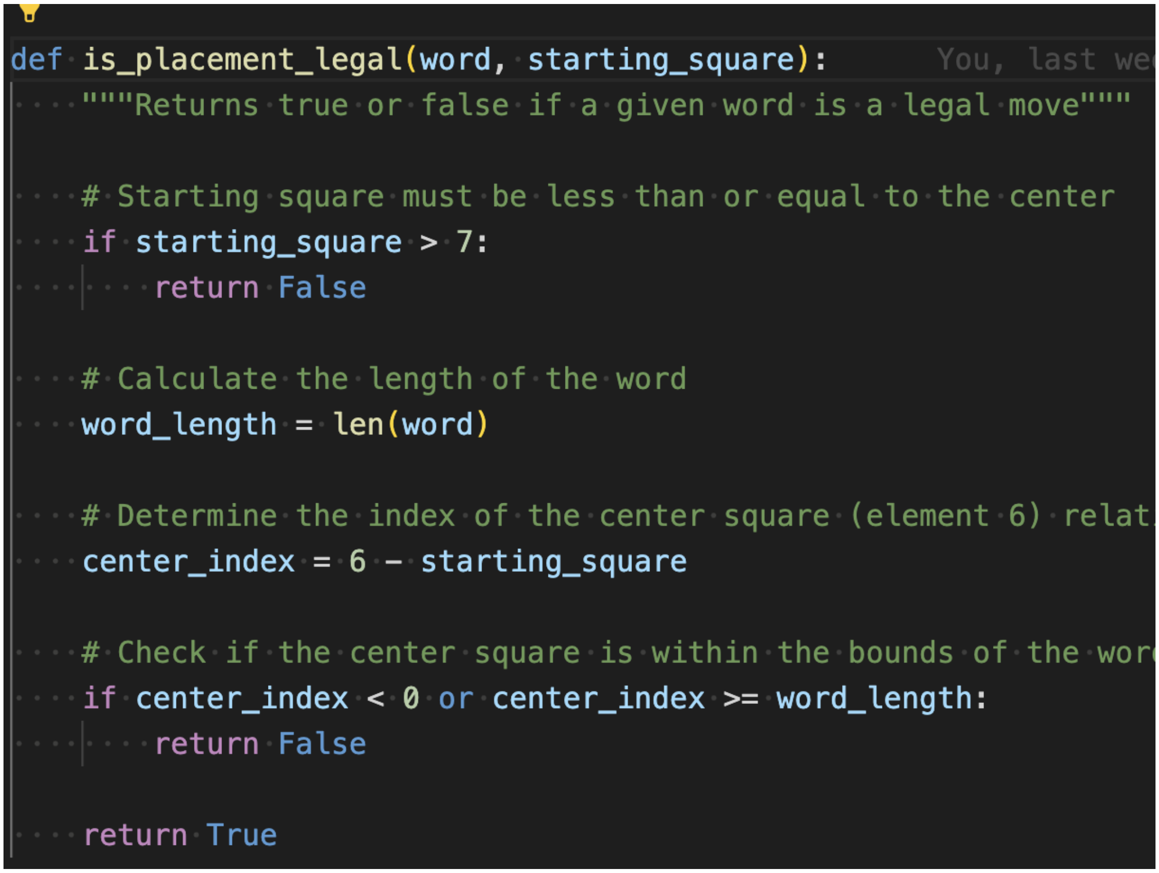
The calculator will calculate a given valid move and some utility functions, like `is_placement_legal`

So I think we should add the dictionary as part of the program. Using an API would be cool and we could even try to reduce the number of API calls as a metric but perhaps we shoot for the moon instead of the stars here.

Now that we have `/data/dictionary.py` included in our project we can make a successor module that checks if a move if valid (for now). Here are the rules of a first move in Scrabble:

- It must be a valid word from the dictionary
- It must be at least 2 tiles
- It must cover the center square
- It must be placed horizontally

We can write some heuristics into the successor module as well; for example we can make sure it's checking from left to right (as thats English!), and make sure we are using the most efficient look-up algorithmn as possible to check the dictionary (it's ordered!). So the `is_placement_legal` function will expect a valid word from the dictionary because lets assume our anagram module uses the dictionary:



```
def is_placement_legal(word, starting_square):       You, last we
    """Returns true or false if a given word is a legal move"""

    # Starting square must be less than or equal to the center
    if starting_square > 7:
        return False

    # Calculate the length of the word
    word_length = len(word)

    # Determine the index of the center square (element 6) relat
    center_index = 6 - starting_square

    # Check if the center square is within the bounds of the wor
    if center_index < 0 or center_index >= word_length:
        return False

    return True
```

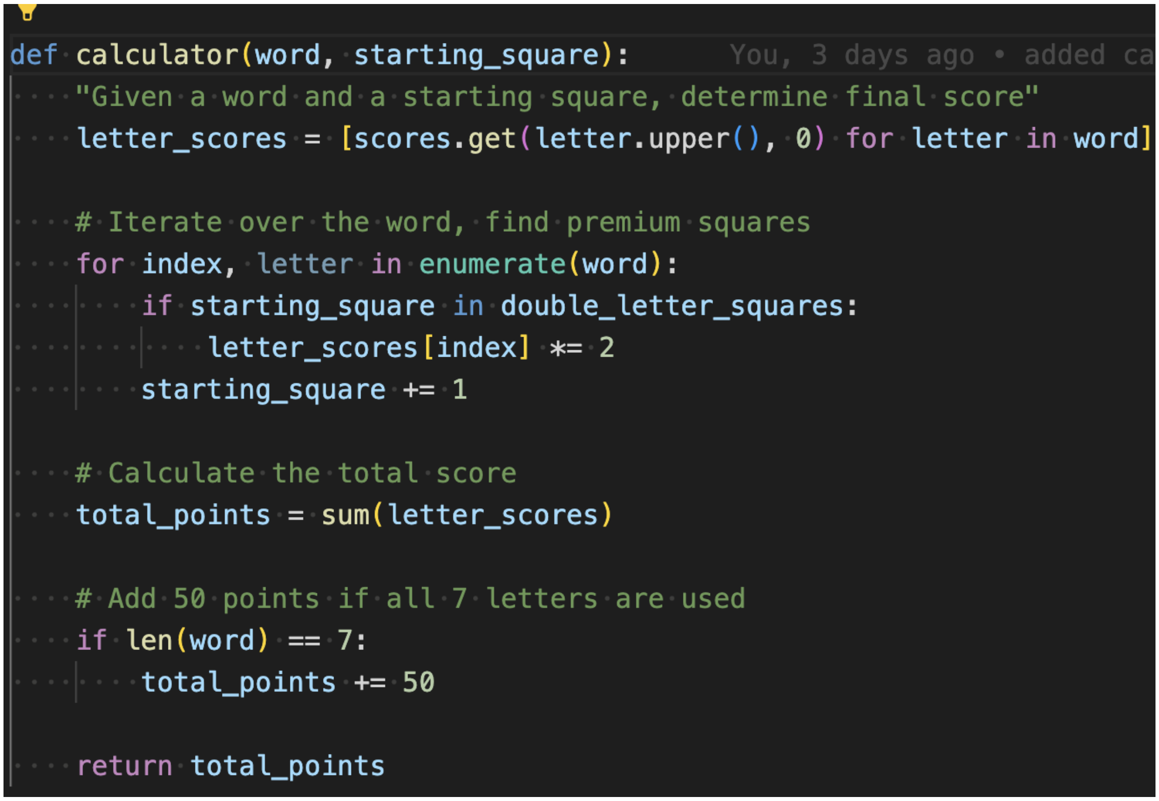
As you can see some heurstics are already in place. We are assuming a word that starts left-to-right, so if the `starter_square` is greater than 7 we can be sure the `center_square` will not be covered as per the rules. If the word starts to the left of the `center_square` it must be at least long enough to cover the `center_square`.

Lets start with the calculator. Pretty straightforward, here how it is scored:

- A, E, I, O, U, L, N, R, S, T = 1 point
- D, G = 2 points
- B, C, M, P = 3 points
- F, H, V, W, Y = 4 points
- J, Q, X = 8 points
- K = 10 points

- Z = 10 points
- There are two double letter score squares, each 4 positions away from the center.
- An additional 50 points for using all 7 letters in the rack

The first thing we need is a dict with the letters and their point values. We will put that into the `points.py` module. We are going to assume that the board state is legal when the `calculator` gets it, so we can use the same argument as the `is_placement_legal` function:

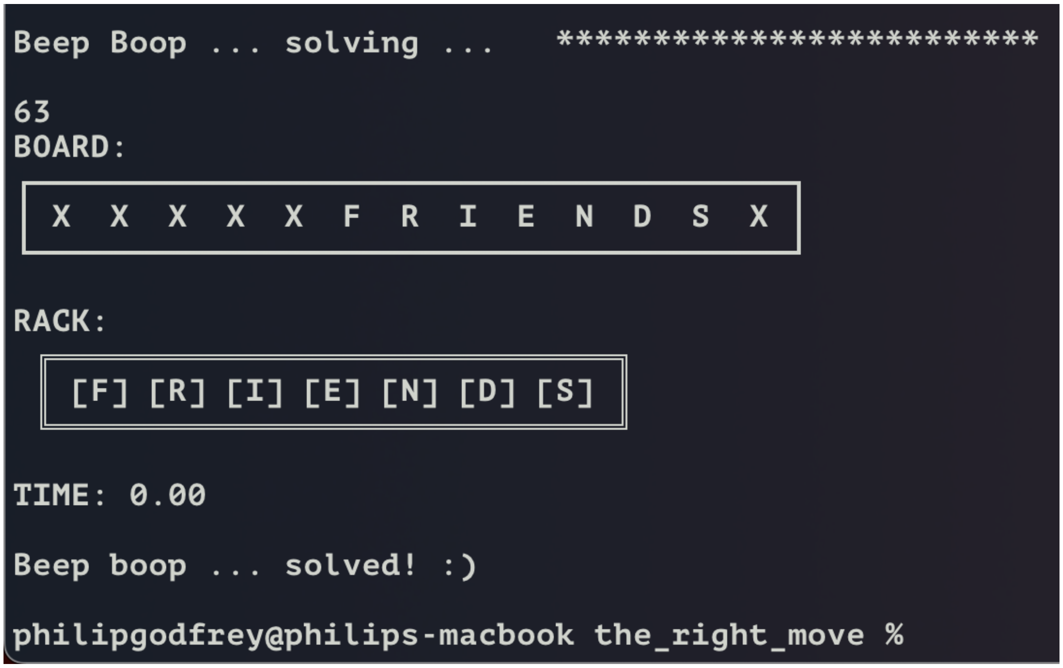


```python
def calculator(word, starting_square):       You, 3 days ago • added ca
    "Given a word and a starting square, determine final score"
    letter_scores = [scores.get(letter.upper(), 0) for letter in word]

    # Iterate over the word, find premium squares
    for index, letter in enumerate(word):
        if starting_square in double_letter_squares:
            letter_scores[index] *= 2
        starting_square += 1

    # Calculate the total score
    total_points = sum(letter_scores)

    # Add 50 points if all 7 letters are used
    if len(word) == 7:
        total_points += 50

    return total_points
```

Pretty clever, put all the point values in an array, double it if its needs to be, then sum it up.

Add 50 if its all 7 letters

Okay cool, so now we have a function that will check if the placement is legal and we have a function that will calculate the score. Lets try them out:



```
Beep Boop ... solving ...   *************************

63
BOARD:
 X  X  X  X  X  F  R  I  E  N  D  S  X

RACK:
  [F] [R] [I] [E] [N] [D] [S]

TIME: 0.00

Beep boop ... solved! :)

philipgodfrey@philips-macbook the_right_move %
```

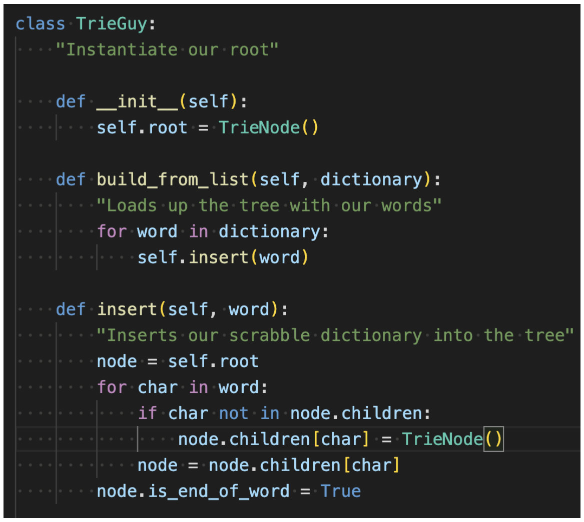
I am fairly certain that is correct. FRIENDS by itself is 11 points, + 2 more for the D being double = 13 + 50 points for using all 7 letters and that is 63 points!

So that is the MAIN app. To make this work at the lowest level the only thing we really need to add in is an anagram creator. Once we have valid words to test for legality, we can figure out how to make it more efficient.

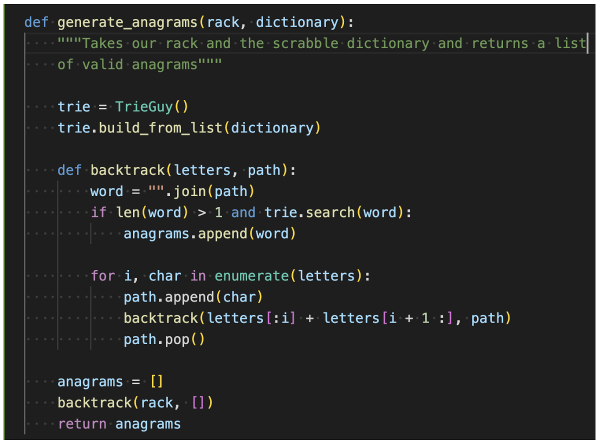
An anagram generator isn't a novel thing. The data structure I am going to use is called a `trie tree` which is a pretty common way to search for strings. It essentially organizes the scrabble dictionary into trees with common nodes, so you don't have to search the entire dictionary. It traverses the tree and when a word is found in the dictionary the flag is triggered and we have a good word.

So we take each letter from the rack, find the root node and follow it until we've got a word. We check if its a good word and boom, its added to our list of of anagrams.

There are two peices, my `TrieGuy` class and the actual `generate_anagrams` function:

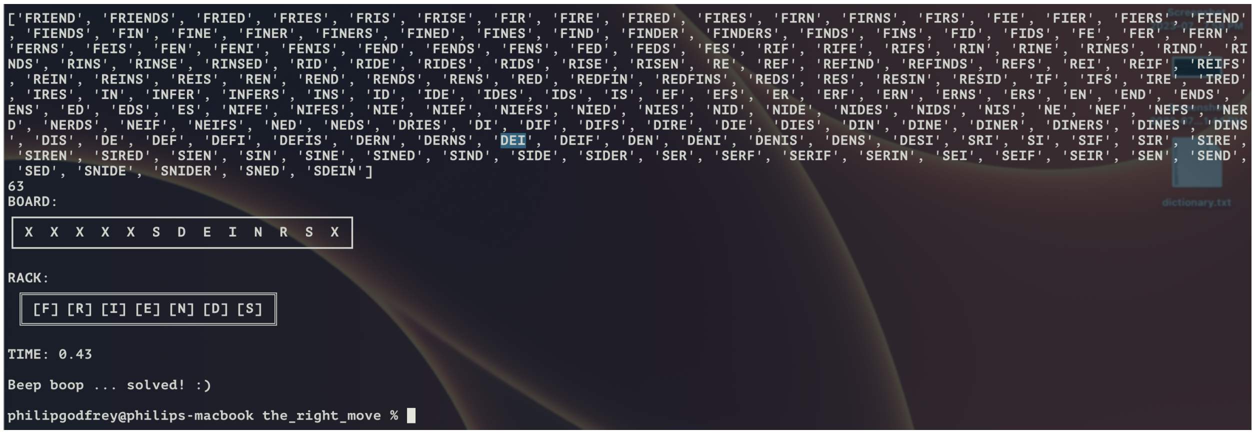


```python
class TrieGuy:
    "Instantiate our root"

    def __init__(self):
        self.root = TrieNode()

    def build_from_list(self, dictionary):
        "Loads up the tree with our words"
        for word in dictionary:
            self.insert(word)

    def insert(self, word):
        "Inserts our scrabble dictionary into the tree"
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
```



```python
def generate_anagrams(rack, dictionary):
    """Takes our rack and the scrabble dictionary and returns a list
    of valid anagrams"""

    trie = TrieGuy()
    trie.build_from_list(dictionary)

    def backtrack(letters, path):
        word = "".join(path)
        if len(word) > 1 and trie.search(word):
            anagrams.append(word)

        for i, char in enumerate(letters):
            path.append(char)
            backtrack(letters[:i] + letters[i + 1 :], path)
            path.pop()

    anagrams = []
    backtrack(rack, [])
    return anagrams
```

I set it up to just try the FRIENDS rack so I could see the output more clearly. This is what I've got:



```
['FRIEND', 'FRIENDS', 'FRIED', 'FRIES', 'FRIS', 'FRISE', 'FIR', 'FIRE', 'FIRED', 'FIRES', 'FIRN', 'FIRNS', 'FIRS', 'FIE', 'FIER', 'FIERS', 'FIEND'
, 'FIENDS', 'FIN', 'FINE', 'FINER', 'FINERS', 'FINED', 'FINES', 'FIND', 'FINDER', 'FINDERS', 'FINDS', 'FINS', 'FID', 'FIDS', 'FE', 'FER', 'FERN',
'FERNS', 'FEIS', 'FEN', 'FENI', 'FENIS', 'FEND', 'FENDS', 'FENS', 'FED', 'FEDS', 'FES', 'RIF', 'RIFE', 'RIFS', 'RIN', 'RINE', 'RINES', 'RIND', 'RI
NDS', 'RINS', 'RINSE', 'RINSED', 'RID', 'RIDE', 'RIDES', 'RIDS', 'RISE', 'RISEN', 'RE', 'REF', 'REFIND', 'REFINDS', 'REFS', 'REI', 'REIF', 'REIFS'
, 'REIN', 'REINS', 'REIS', 'REN', 'REND', 'RENDS', 'RENS', 'RED', 'REDFIN', 'REDFINS', 'REDS', 'RES', 'RESIN', 'RESID', 'IF', 'IFS', 'IRE', 'IRED'
, 'IRES', 'IN', 'INFER', 'INFERS', 'INS', 'ID', 'IDE', 'IDES', 'IDS', 'IS', 'EF', 'EFS', 'ER', 'ERF', 'ERN', 'ERNS', 'ERS', 'EN', 'END', 'ENDS', '
ENS', 'ED', 'EDS', 'ES', 'NIFE', 'NIFES', 'NIE', 'NIEF', 'NIEFS', 'NIED', 'NIES', 'NID', 'NIDE', 'NIDES', 'NIDS', 'NIS', 'NE', 'NEF', 'NEFS', 'NER
D', 'NERDS', 'NEIF', 'NEIFS', 'NED', 'NEDS', 'DRIES', 'DI', 'DIF', 'DIFS', 'DIRE', 'DIE', 'DIES', 'DIN', 'DINE', 'DINER', 'DINERS', 'DINES', 'DINS
', 'DIS', 'DE', 'DEF', 'DEFI', 'DEFIS', 'DERN', 'DERNS', 'DEI', 'DEIF', 'DEN', 'DENI', 'DENIS', 'DENS', 'DESI', 'SRI', 'SI', 'SIF', 'SIR', 'SIRE',
 'SIREN', 'SIRED', 'SIEN', 'SIN', 'SINE', 'SINED', 'SIND', 'SIDE', 'SIDER', 'SER', 'SERF', 'SERIF', 'SERIN', 'SEI', 'SEIF', 'SEIR', 'SEN', 'SEND',
 'SED', 'SNIDE', 'SNIDER', 'SNED', 'SDEIN']
63
BOARD:
┌─────────────────────────────┐
│ X  X  X  X  X  S  D  E  I  N  R  S  X │
└─────────────────────────────┘

RACK:
 ┌──────────────────────────────┐
 │ [F] [R] [I] [E] [N] [D] [S] │
 └──────────────────────────────┘

TIME: 0.43

Beep boop ... solved! :)

philipgodfrey@philips-macbook the_right_move % █
```

Awesome! It's only outputting the board after it places the last word, and we are automatically starting on 5, so thats why SDEINRS is the last word (I think its adding the left over letters).

One last thing we need to add: handling blank letters (`_`). This can be ANY letter but iterating over the whole alphabet and trying each one is inefficient because there are some letters more common than others, and some letters that are worth more points than others.

Now we are getting into the heuristics. Here are the letters of the alphabet in order of most commonly used (in English):

```
etaoinshrdlcumwfgypbvkjxqz
```

I think we should re-order this with the weight of the scores included. For example, Z is 10 points, but it appears last in our list. The goal of the agent is the highest score possible so this is how we are are going to weight the letters: `.6 for score, .5 for commonality`

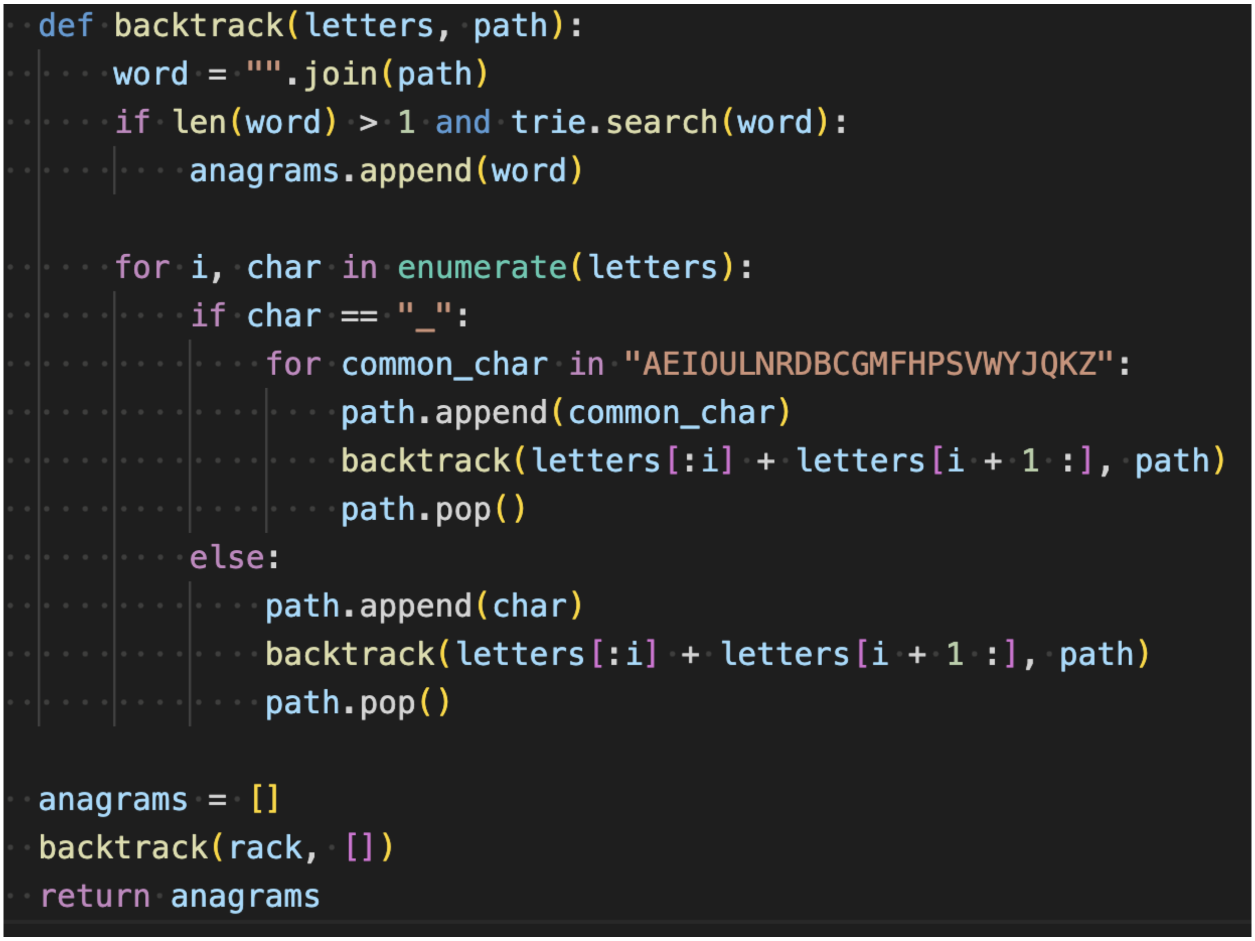
I wrote a script to provide me with the weighted scores of each letter, giving Score a 0.6 and Commonality a 0.4, so scores are weighted a little more than commonality:

```
weighted_scores = {}
for char in letter_scores:
  letter_score = letter_scores[char]
  commonality_score = (
  commonality_ranking.index(char) / len(commonality_ranking)
  if char in commonality_ranking
  else 0
)
weighted_scores[char] = 0.6 * (letter_score / 10) + 0.4 * commonality_score
```

and this gives me the order I'll use to pick the blank tile:

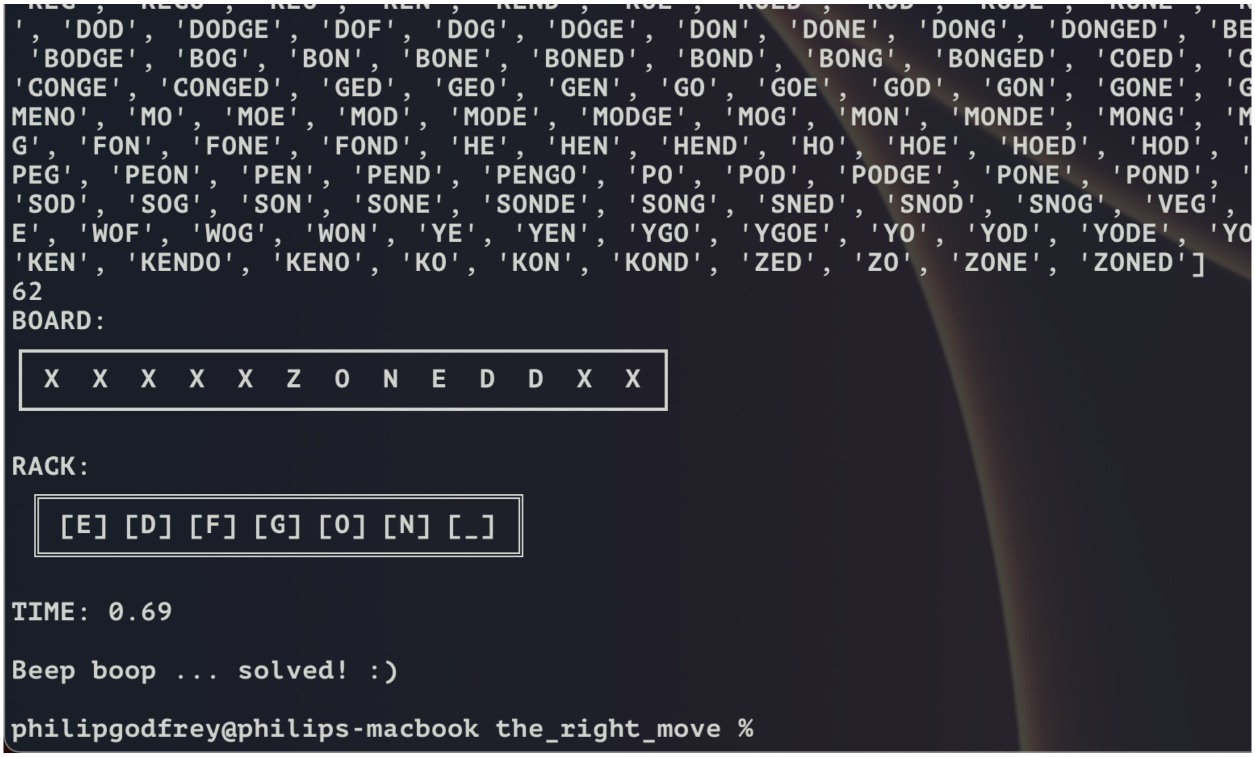
```
AEIOULNRDBCGMFHPSVWYJQKZ
```

Vowels are first! Thats pretty cool and makes sense I think. It makes less sense if the preceding letter is also a vowel but that is another huerstic I am not going to get into right now! Lets test the code:



```python
    def backtrack(letters, path):
        word = "".join(path)
        if len(word) > 1 and trie.search(word):
            anagrams.append(word)

        for i, char in enumerate(letters):
            if char == "_":
                for common_char in "AEIOULNRDBCGMFHPSVWYJQKZ":
                    path.append(common_char)
                    backtrack(letters[:i] + letters[i + 1 :], path)
                    path.pop()
            else:
                path.append(char)
                backtrack(letters[:i] + letters[i + 1 :], path)
                path.pop()

    anagrams = []
    backtrack(rack, [])
    return anagrams
```

This is the bottom of the anagram list. As you can see the blank tile represented as a Z is at the bottom.

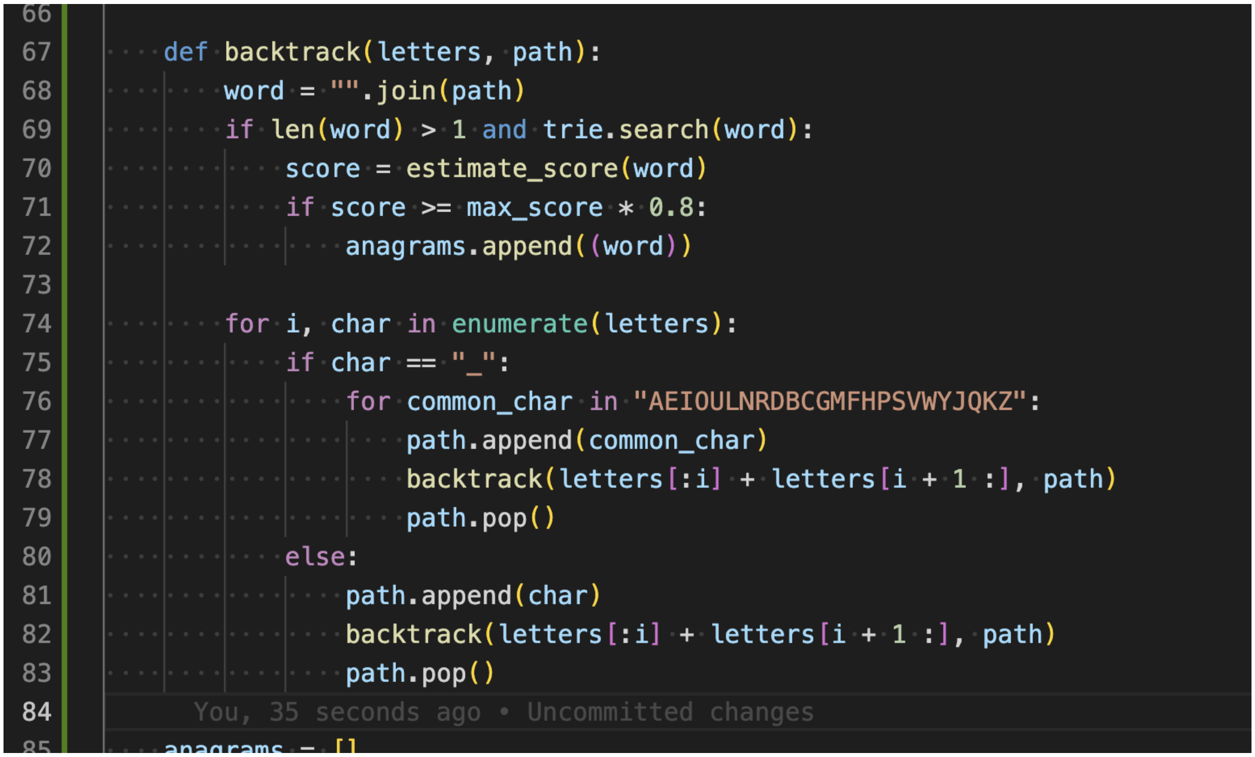


```
', 'DOD', 'DODGE', 'DOF', 'DOG', 'DOGE', 'DON', 'DONE', 'DONG', 'DONGED', 'BE
 'BODGE', 'BOG', 'BON', 'BONE', 'BONED', 'BOND', 'BONG', 'BONGED', 'COED', 'C
'CONGE', 'CONGED', 'GED', 'GEO', 'GEN', 'GO', 'GOE', 'GOD', 'GON', 'GONE', 'G
MENO', 'MO', 'MOE', 'MOD', 'MODE', 'MODGE', 'MOG', 'MON', 'MONDE', 'MONG', 'M
G', 'FON', 'FONE', 'FOND', 'HE', 'HEN', 'HEND', 'HO', 'HOE', 'HOED', 'HOD', '
PEG', 'PEON', 'PEN', 'PEND', 'PENGO', 'PO', 'POD', 'PODGE', 'PONE', 'POND', '
'SOD', 'SOG', 'SON', 'SONE', 'SONDE', 'SONG', 'SNED', 'SNOD', 'SNOG', 'VEG',
E', 'WOF', 'WOG', 'WON', 'YE', 'YEN', 'YGO', 'YGOE', 'YO', 'YOD', 'YODE', 'YO
'KEN', 'KENDO', 'KENO', 'KO', 'KON', 'KOND', 'ZED', 'ZO', 'ZONE', 'ZONED']
62
BOARD:
┌──────────────────────────────────────────┐
│ X  X  X  X  X  Z  O  N  E  D  D  X  X    │
└──────────────────────────────────────────┘

RACK:
  ╔════════════════════════════╗
  ║ [E] [D] [F] [G] [O] [N] [_] ║
  ╚════════════════════════════╝

TIME: 0.69

Beep boop ... solved! :)

philipgodfrey@philips-macbook the_right_move %
```

So now we have a pretty full anagram generator, lets try to reduce the size of the list. We don’t want to have to go through and calculate every single one of these anagrams, so lets set a threshold:

- estimate max score of rack
- only add words that are in the 20th percentile of the score

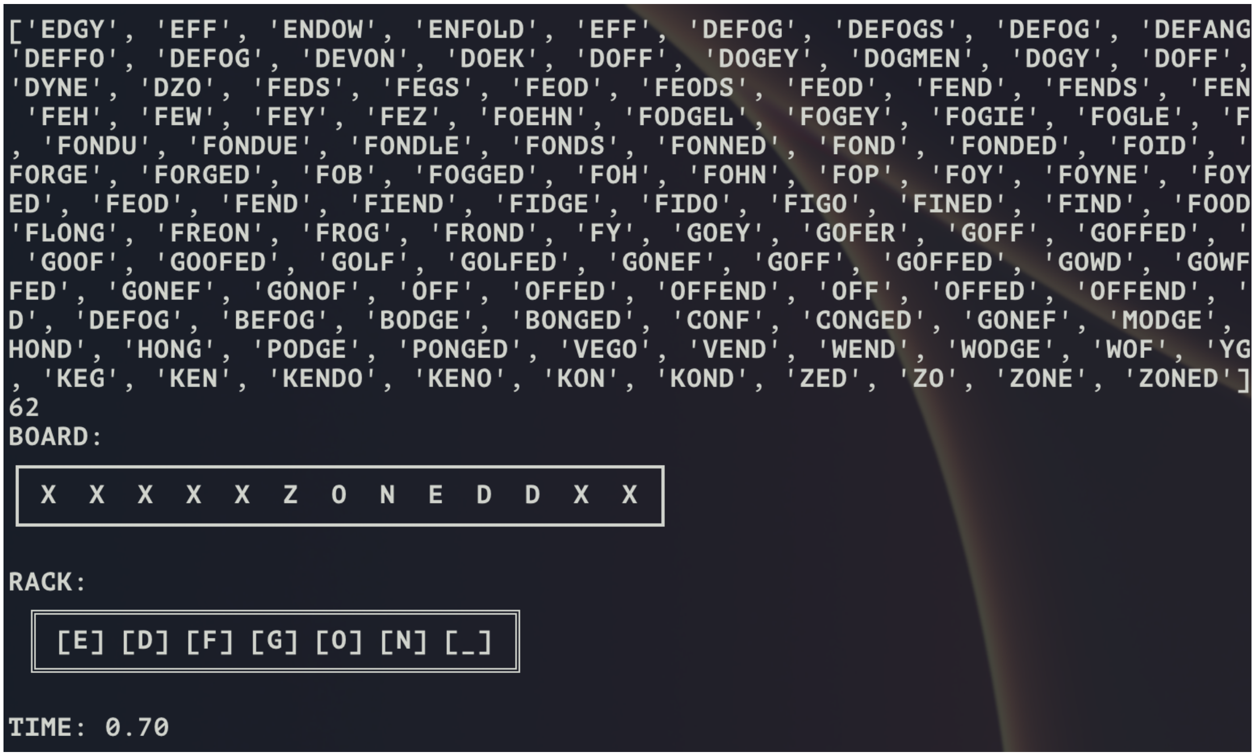
My estimation method will add all the letters and since the first move can only cover 1 double letter score, we double the tile with the largest score:



```
    def backtrack(letters, path):
        word = "".join(path)
        if len(word) > 1 and trie.search(word):
            score = estimate_score(word)
            if score >= max_score * 0.8:
                anagrams.append((word))

        for i, char in enumerate(letters):
            if char == "_":
                for common_char in "AEIOULNRDBCGMFHPSVWYJQKZ":
                    path.append(common_char)
                    backtrack(letters[:i] + letters[i + 1 :], path)
                    path.pop()
            else:
                path.append(char)
                backtrack(letters[:i] + letters[i + 1 :], path)
                path.pop()
          You, 35 seconds ago • Uncommitted changes
    anagrams = []
```

The reason we aren’t adding the bonus points for a rack with a blank tile is that we cannot gurantee there will be any valid anagrams of 7 letters. So we end up with no anagrams in our list.

All and all this significantly reduces our `anagrams`



```
['EDGY', 'EFF', 'ENDOW', 'ENFOLD', 'EFF', 'DEFOG', 'DEFOGS', 'DEFOG', 'DEFANG
'DEFFO', 'DEFOG', 'DEVON', 'DOEK', 'DOFF', 'DOGEY', 'DOGMEN', 'DOGY', 'DOFF',
'DYNE', 'DZO', 'FEDS', 'FEGS', 'FEOD', 'FEODS', 'FEOD', 'FEND', 'FENDS', 'FEN
 'FEH', 'FEW', 'FEY', 'FEZ', 'FOEHN', 'FODGEL', 'FOGEY', 'FOGIE', 'FOGLE', 'F
, 'FONDU', 'FONDUE', 'FONDLE', 'FONDS', 'FONNED', 'FOND', 'FONDED', 'FOID', '
FORGE', 'FORGED', 'FOB', 'FOGGED', 'FOH', 'FOHN', 'FOP', 'FOY', 'FOYNE', 'FOY
ED', 'FEOD', 'FEND', 'FIEND', 'FIDGE', 'FIDO', 'FIGO', 'FINED', 'FIND', 'FOOD
'FLONG', 'FREON', 'FROG', 'FROND', 'FY', 'GOEY', 'GOFER', 'GOFF', 'GOFFED', '
 'GOOF', 'GOOFED', 'GOLF', 'GOLFED', 'GONEF', 'GOFF', 'GOFFED', 'GOWD', 'GOWF
FED', 'GONEF', 'GONOF', 'OFF', 'OFFED', 'OFFEND', 'OFF', 'OFFED', 'OFFEND', '
D', 'DEFOG', 'BEFOG', 'BODGE', 'BONGED', 'CONF', 'CONGED', 'GONEF', 'MODGE',
HOND', 'HONG', 'PODGE', 'PONGED', 'VEGO', 'VEND', 'WEND', 'WODGE', 'WOF', 'YG
, 'KEG', 'KEN', 'KENDO', 'KENO', 'KON', 'KOND', 'ZED', 'ZO', 'ZONE', 'ZONED']
62
BOARD:

 X  X  X  X  X  Z  O  N  E  D  D  X  X

RACK:

  [E] [D] [F] [G] [O] [N] [_]

TIME: 0.70
```

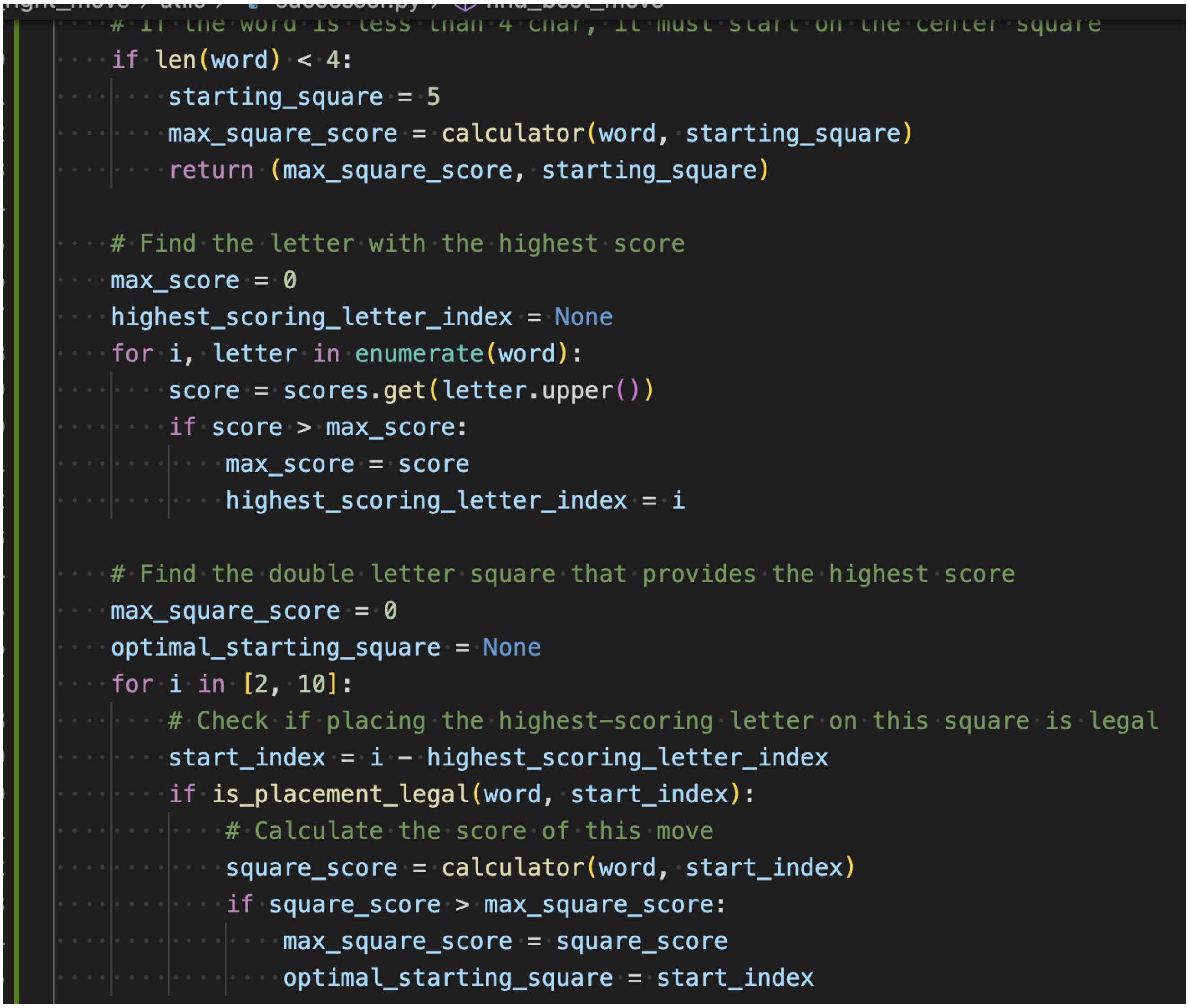
Outstanding. So lets sum up what we have done so far:

- Scrabble Board representing the middle row (horizontal)
- Agent that reads in a `rack` can place tiles on the squares and can calculate the score
- Agent contains `solve()` method that utilizes our `successor` module:
    - `generate_anagrams` will take the rack and generate anagrams, if a `_` blank tile is found, the choice of letters is ordered by 60% score and 40% commonality.
    - it will also `estimate_score` of the rack, and only add words to the list if they are in the 20th percentile of the `estimate_score` .
- Agent then places the last word in the list and returns the score.

The very last thing to do is pick the winner. We could sort the anagrams list by estimated score but I think thats probably a waste of time. At this point the `anagrams` list isn’t that long, its probably faster to just place and calculate.

So now we need a function to decide the starting square when we place the word. Since we are going with highest score possible this is the order we should use to decide:

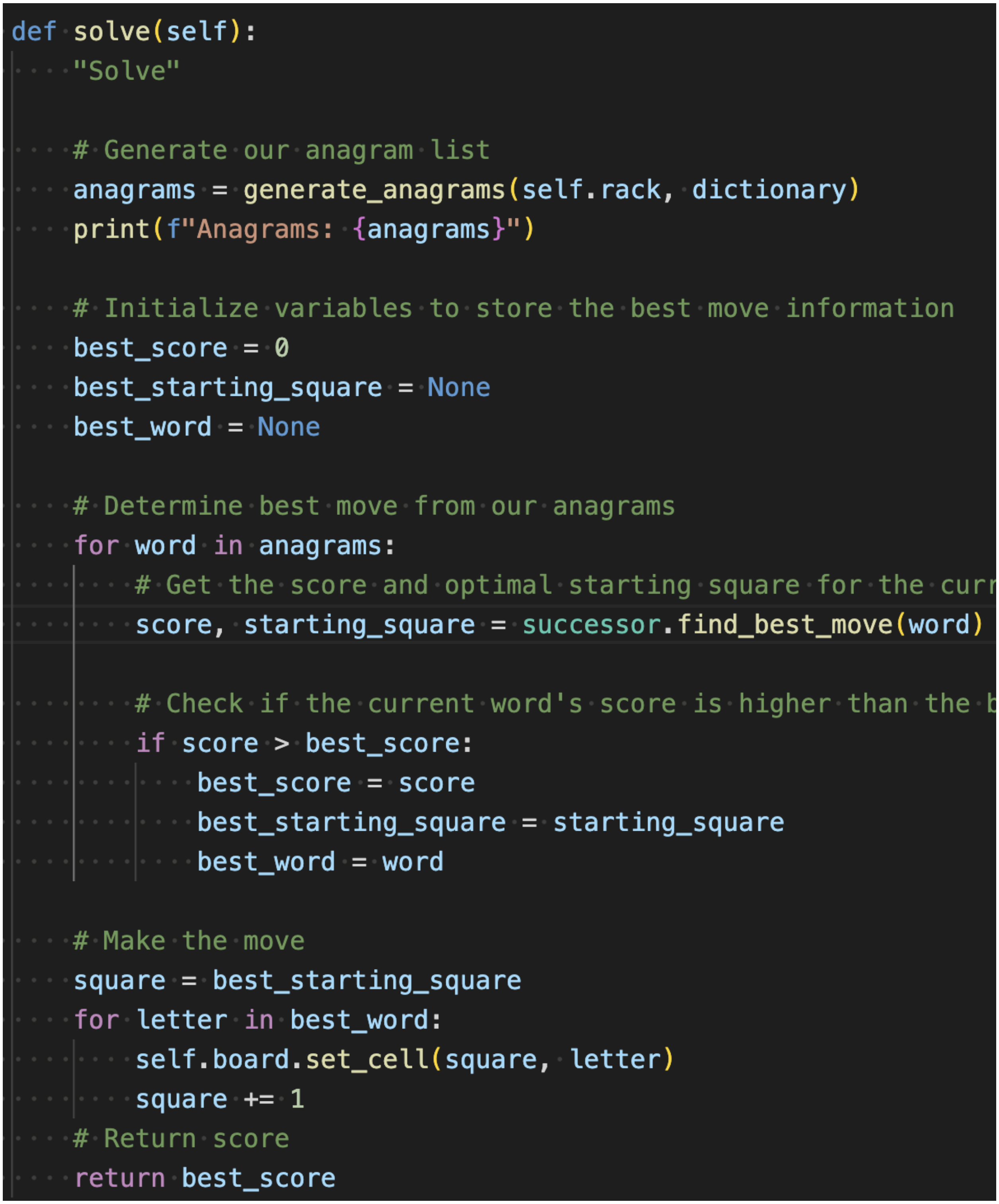
- letter with highest score goes on whatever double letter square provides:
    1. a legal move
    2. the higher score
- If word is less than 4 characters, it must start on the center square



```python
    # If the word is less than 4 char, it must start on the center square
    if len(word) < 4:
        starting_square = 5
        max_square_score = calculator(word, starting_square)
        return (max_square_score, starting_square)

    # Find the letter with the highest score
    max_score = 0
    highest_scoring_letter_index = None
    for i, letter in enumerate(word):
        score = scores.get(letter.upper())
        if score > max_score:
            max_score = score
            highest_scoring_letter_index = i

    # Find the double letter square that provides the highest score
    max_square_score = 0
    optimal_starting_square = None
    for i in [2, 10]:
        # Check if placing the highest-scoring letter on this square is legal
        start_index = i - highest_scoring_letter_index
        if is_placement_legal(word, start_index):
            # Calculate the score of this move
            square_score = calculator(word, start_index)
            if square_score > max_square_score:
                max_square_score = square_score
                optimal_starting_square = start_index
```

So we are essentially taking the word, trying to put it on the board such that the highest scoring letter is doubled. We do this over and over until we have the best place to put each word.

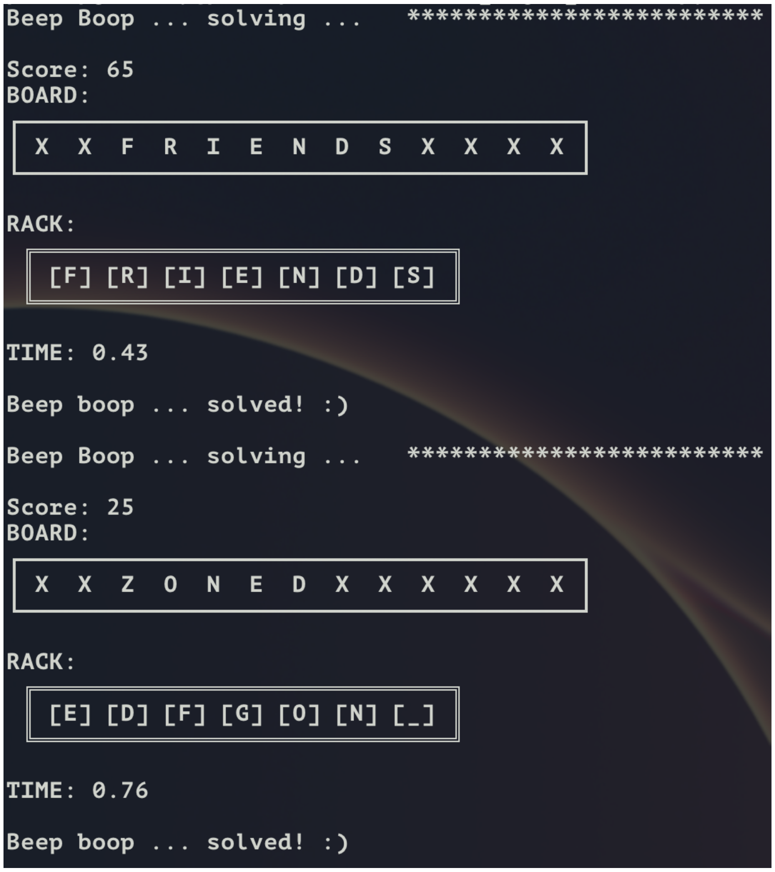
So now my agent solves like this:



```python
def solve(self):
    "Solve"

    # Generate our anagram list
    anagrams = generate_anagrams(self.rack, dictionary)
    print(f"Anagrams: {anagrams}")

    # Initialize variables to store the best move information
    best_score = 0
    best_starting_square = None
    best_word = None

    # Determine best move from our anagrams
    for word in anagrams:
        # Get the score and optimal starting square for the curr
        score, starting_square = successor.find_best_move(word)

        # Check if the current word's score is higher than the b
        if score > best_score:
            best_score = score
            best_starting_square = starting_square
            best_word = word

    # Make the move
    square = best_starting_square
    for letter in best_word:
        self.board.set_cell(square, letter)
        square += 1
    # Return score
    return best_score
```

- Generate anagrams
- Find highest score out of all of the anagrams
- Place the word
- Display

This works!



```
Beep Boop ... solving ...   ************************

Score: 65
BOARD:

 X  X  F  R  I  E  N  D  S  X  X  X  X

RACK:

  [F] [R] [I] [E] [N] [D] [S]

TIME: 0.43

Beep boop ... solved! :)

Beep Boop ... solving ...   ************************

Score: 25
BOARD:

 X  X  Z  O  N  E  D  X  X  X  X  X  X

RACK:

  [E] [D] [F] [G] [O] [N] [_]

TIME: 0.76

Beep boop ... solved! :)
```

So the 2nd rack is a good one to test because it has a blank tile. Turns out there are NO valid anagrams with those letters including the blank tile that are 7 characters in length. SO if we were to add 50 to the estimate we would never hit the estimate.

It's possible this will bite us, we could come up with an anagram list with some 7 letter words that when the bonus is added is more than a 6 letter word.

The solution here would be to make a more robust estimator or just replace it altogether with a call to `find_best_move` and `set_cell` .

Or remove the 50 bonus points, that would make this scrabble solver VERY accurate.