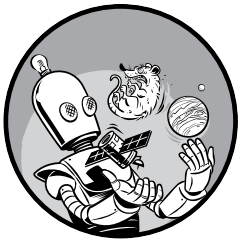
3

SOLVING ANAGRAMS



An *anagram* is a word formed by rearranging the letters of another word. For example, *Elvis* yields the eerie trio *evils*, *lives*, and *veils*. Does this mean Elvis still lives but veils his evil existence? In the book *Harry Potter and the Chamber of Secrets*, "I am Lord Voldemort" is an anagram of the evil wizard's real name, Tom Marvolo Riddle. "Lord Earldom Vomit" is also an anagram of Tom Marvolo Riddle, but author J.K. Rowling had the good sense to pass on that one.

In this chapter, first you'll find all the anagrams for a given word or name. Then, you'll write a program that lets a user interactively build an anagram phrase from their own name. Finally, you'll play computer wizard and see what it takes to extract "I am Lord Voldemort" from "Tom Marvolo Riddle."

Project #4: Finding Single-Word Anagrams

You'll start by analyzing simple single-word anagrams and figuring out how to identify them programmatically. Having accomplished this, you'll be ready to take on anagram phrases in the following section.

THE OBJECTIVE

Use Python and a dictionary file to find all the single-word anagrams for a given English word or single name. You can read instructions for finding and loading dictionary files at the start of Chapter 2.

The Strategy and Pseudocode

More than 600 newspapers and 100 internet sites carry an anagram game called *Jumble*. Created in 1954, it's now the most recognized word-scramble game in the world. *Jumble* can be really frustrating, but finding anagrams is almost as easy as finding palindromes—you just need to know the common characteristic of all anagrams: they must have the same number of the same letters.

Identifying an Anagram

Python doesn't contain a built-in anagram operator, but you can easily write one. For the projects in this chapter, you'll load the dictionary file from Chapter 2 as a list of strings. So the program needs to verify that two strings are anagrams of each other.

Let's look at an example. *Pots* is an anagram of *stop*, and you can verify that *stop* and *pots* have the same number of letters with the `len()` function. But there's no way for Python to know whether two strings have the same number of any single character—at least not without converting the strings to another data structure or using a counting function. So, instead of looking at these two words simply as strings, you can represent them as two lists containing single-character strings. Create these lists in a shell, like IDLE, and name them `word` and `anagram`, as I've done here:

```
>>> word = list('stop')
>>> word
['s', 't', 'o', 'p']
>>> anagram = list('pots')
>>> anagram
['p', 'o', 't', 's']
```

These two lists match our description of an anagram pair; that is, they contain the same number of the same letters. But if you try to equate them with the comparison operator `==`, the result is `False`.

```
>>> anagram == word
False
```

The problem is that the operator (==) considers two lists equivalent only if they have the same number of the same list items and those items occur in the same order. You can easily solve this problem with the built-in function `sorted()`, which can take a list as an argument and reorder its contents alphabetically. So, if you call `sorted()` twice—once for each of the lists—and then compare the sorted lists, they will be equivalent. In other words, `==` returns `True`.

```
>>> word = sorted(word)
>>> word
['o', 'p', 's', 't']
>>> anagram = sorted(anagram)
>>> anagram
['o', 'p', 's', 't']
>>> anagram == word
True
```

You can also pass a string to `sorted()` to create a sorted list like the ones in the preceding code snippet. This will be useful for converting the words from the dictionary file into sorted lists of single-character strings.

Now that you know how to verify that you've found an anagram, let's design the script in its entirety—from loading a dictionary and prompting the user for a word (or name) to searching for and printing all the anagrams.

Using Pseudocode

Remember that planning with pseudocode will help you spot potential issues and spotting those issues early will save you time. The following pseudocode should help you better understand the script we'll write in the next section, *anagrams.py*.

```
Load digital dictionary file as a list of words
Accept a word from user
Create an empty list to hold anagrams
Sort the user-word
Loop through each word in the word list:
    Sort the word
    if word sorted is equal to user-word sorted:
        Append word to anagrams list
Print anagrams list
```

The script will start by loading words from a dictionary file into a list as strings. Before you loop through the dictionary in search of anagrams, you need to know which word you want anagrams of, and you need a place to store anagrams when you find them. So, first ask the user to input a word

and then create an empty list to store the anagrams. Once the program has looped through every word in the dictionary, it will print that list of anagrams.

Anagram-Finder Code

Listing 3-1 loads a dictionary file, accepts a word or name *specified within the program*, and finds all the anagrams in the dictionary file for that word or name. You'll also need the dictionary-loading code from Chapter 2. You can download these from *https://www.nostarch.com/impracticalpython/* as *anagrams.py* and *load_dictionary.py*, respectively. Keep both files in the same folder. You can use the same dictionary file you used in Chapter 2 or download another one (see Table 2-1 on page 20 for suggestions).

anagrams.py

```
❶ import load_dictionary

❷ word_list = load_dictionary.load('2of4brif.txt')

❸ anagram_list = []

  # input a SINGLE word or SINGLE name below to find its anagram(s):
❹ name = 'Foster'
  print("Input name = {}".format (name))
❺ name = name.lower()
  print("Using name = {}".format(name))

  # sort name & find anagrams
❻ name_sorted = sorted(name)
❼ for word in word_list:
      word = word.lower()
      if word != name:
          if sorted(word) == name_sorted:
              anagram_list.append(word)

  # print out list of anagrams
  print()
❽ if len(anagram_list) == 0:
      print("You need a larger dictionary or a new name!")
  else:
   ❾ print("Anagrams =", *anagram_list, sep='\n')
```

Listing 3-1: Given a word (or name) and a dictionary file, this program searches for and prints a list of anagrams.

You start by importing the `load_dictionary` module you created in Chapter 2 ❶. This module will open a dictionary text file and, with its `load()` function, load all the words into a list ❷. The **.txt* file you use may be different, depending on which dictionary file you downloaded (see "Finding and Opening a Dictionary" on page 20).

Next, create an empty list, called `anagram_list`, to hold any anagrams you find ❸. Have the user add a *single* word, such as their first name ❹. This

doesn't have to be a proper name, but we'll refer to it as `name` in the code to distinguish it from a dictionary `word`. Print this name so the user can see what was entered.

The next line anticipates a problematic user action. People tend to type their name with an uppercase first letter, but dictionary files may not include uppercase letters, and that matters to Python. So, first convert all letters to lowercase with the `.lower()`string method ❺.

Now sort the name ❻. As mentioned previously, you can pass `sorted()` a string as well as a list.

With the input sorted alphabetically in a list, it's time to find anagrams. Start a loop through each word in the dictionary word list ❼. To be safe, convert the word to lowercase, as comparison operations are case-sensitive. After the conversion, compare the word to the unsorted name, because a word can't be an anagram of itself. Next, sort the dictionary word and compare it to the sorted name. If it passes, append that dictionary word to `anagram_list`.

Now display the results. First, check whether the anagram list is empty. If it is, print a whimsical reply so you don't just leave the user hanging ❽. If the program found at least one anagram, print the list using the splat (*) operator. Remember from Chapter 2 that splat lets you print each member of a list on a separate line ❾.

The following is example output for this program, using the input name *Foster*:

```
Input name = Foster
Using name = foster

Anagrams =
forest
fortes
softer
```

If you'd like to use another input, change the value of the `name` variable in the source code. As an exercise, try to adjust the code so that the user is prompted to input the name (or word); you can do this with the `input()` function.

Project #5: Finding Phrase Anagrams

In the previous project, you took a single name or word and rearranged all the letters to find single-word anagrams. Now you will derive multiple words from a name. The words in these *phrase anagrams* form only part of the input name, and you will need several words to exhaust the available letters.

THE OBJECTIVE

Write a Python program that lets a user interactively build an anagram phrase from the letters in their name.

The Strategy and Pseudocode

The very best phrase anagrams are those that describe some well-known characteristic or action associated with the name bearer. For example, the letters in Clint Eastwood can be rearranged to form *old west action*, Alec Guinness yields *genuine class*, Madam Curie produces *radium came*, George Bush gives *he bugs Gore*, and Statue of Liberty contains *built to stay free*. My own name yields *a huge navel*, which is not really one of my characteristics.

At this point, you may see a strategic challenge ahead: how does a computer handle contextual content? The folks at IBM who invented Watson seem to know, but for the rest of us, that boulder is a little hard to lift.

The *brute-force method* is a common approach used in online anagram generators. These algorithms take a name and return lots of random anagram phrases (generally, 100s to 10,000+). Most of the returned phrases are nonsense, and scrolling through hundreds of these can be a chore.

An alternative approach is to acknowledge that humans are best at contextual issues and write a program that helps the human work through the problem. The computer can take the initial name and provide words that can be made from some (or all) the letters in it; the user can then choose a word that "makes sense." The program will then recalculate the word choices from the remaining letters in the name, repeating the process until every letter is used or the possible word choices are exhausted. This design plays to the strengths of both participants.

You'll need a simple interface that prompts the user to input the initial name, displays potential word choices, and displays any remaining letters. The program will also need to keep track of the growing anagram phrase and let the user know when every letter has been used. There will likely be lots of failed attempts, so the interface should allow the user to restart the process at any time.

Since anagrams have the same number of the same letters, another way to identify them is to count individual letters. If you think of your name as a collection of letters, then a word can be built from your name if (1) all its letters occur in your name and (2) they occur *at the same frequency or less*. Obviously, if *e* occurs three times in a word and twice in your name, the word can't be derived from your name. So, if the collection of letters that make up a word is not a subset of the collection of letters in your name, then that word cannot be part of your name anagram.

Using Counter to Tally Letters

Fortunately for us, Python ships with a module named `collections` that includes several container data types. One of these types, `Counter`, counts the occurrences of an item. Python stores the items as dictionary keys and the counts as dictionary values. For example, the following code snippet counts how many of each bonsai tree type is in a list.

```
>>> from collections import Counter
❶ >>> my_bonsai_trees = ['maple', 'oak', 'elm', 'maple', 'elm', 'elm', 'elm', 'elm']
❷ >>> count = Counter(my_bonsai_trees)
```

```
>>> print(count)
❸ Counter({'elm': 5, 'maple': 2, 'oak': 1})
```

The `my_bonsai_trees` list contains multiples of the same type of tree ❶. `Counter` tallies up the trees ❷ and creates an easy-to-reference dictionary ❸. Note that the `print()` function is optional and is used here for clarity. Entering `count`, alone, will also display the dictionary contents.

You can use `Counter`, instead of the `sorted()` method, to find single-word anagrams. Rather than two sorted lists, the output will be two dictionaries, which can also be directly compared with `==`. Here's an example:

```
>>> name = 'foster'
>>> word = 'forest'
>>> name_count = Counter(name)
>>> print(name_count)
❶ Counter({'f': 1, 't': 1, 'e': 1, 'o': 1, 'r': 1, 's': 1})
>>> word_count = Counter(word)
>>> print(word_count)
❷ Counter({'f': 1, 't': 1, 'o': 1, 'e': 1, 'r': 1, 's': 1})
```

`Counter` produces a dictionary for each word that maps each letter in the word to the number of times it occurs ❶❷. The dictionaries are unsorted, but despite the lack of sorting, Python correctly identifies each dictionary as being equal if the dictionaries contain the same letters and the same counts:

```
>>> if word_count == name_count:
        print("It's a match!")

It's a match!
```

A `Counter` gives you a wonderful way to find words that "fit" in a name. If the count for each letter in a word is less than or equal to the count for the same letter in the name, then the word can be derived from the name!

The Pseudocode

We've now made two important design decisions: (1) let the user interactively build their anagram one word at a time and (2) use the `Counter` method to find anagrams. This is enough to start thinking about high-level pseudocode:

```
Load a dictionary file
Accept a name from user
Set limit = length of name
Start empty list to hold anagram phrase
While length of phrase < limit:
    Generate list of dictionary words that fit in name
    Present words to user
    Present remaining letters to user
    Present current phrase to user
    Ask user to input word or start over
```

```
    If user input can be made from remaining letters:
        Accept choice of new word or words from user
        Remove letters in choice from letters in name
        Return choice and remaining letters in name
    If choice is not a valid selection:
        Ask user for new choice or let user start over
    Add choice to phrase and show to user
    Generate new list of words and repeat process
When phrase length equals limit value:
    Display final phrase
    Ask user to start over or to exit
```

Divvying Up the Work

As procedural code becomes more complex, it becomes necessary to encapsulate much of it in functions. This makes it easier to manage input and output, perform recursion, and read the code.

A *main function* is where a program starts its execution, and enables high-level organization, such as managing all the bits and pieces of the code, including dealing with the user. In the phrase anagram program, the main function will wrap all the "worker bee" functions, take *most* of the user input, keep track of the growing anagram phrase, determine when the phrase is complete, and show the user the result.

Sketching out the tasks and their flow with pencil and paper is a great way to figure out what you want to do and where (like "graphical pseudocode"). Figure 3-1 is a flowchart with function assignments highlighted. In this case, three functions should be sufficient: `main()`, `find_anagrams()`, and `process_choice()`.

The `main()` function's primary task is to set the letter count limit and manage the `while` loop responsible for the general phrase anagram build. The `find_anagrams()` function will take the current collection of letters remaining in a name and return all possible words that can be made from those letters. The words are then displayed for the user, along with the current phrase, which is "owned" and displayed by the `main()` function. Then, the `process_choice()` function prompts the user to start over or choose a word for the anagram phrase. If the user makes a choice, this function determines whether the letters in the choice are available. If they aren't, the user is prompted to choose again or start over. If the user makes a valid choice, the letters in the user's choice are removed from the list of remaining letters, and both the choice and list of leftovers are returned. The `main()` function adds the returned choice to the existing phrase. If the limit is reached, the completed phrase anagram is displayed, and the user is asked to start over or exit.

Note that you ask for the initial name in the *global* scope, rather than in the `main()` function. This allows the user to start over fresh at any time without having to re-enter their name. For now, if the user wants to choose a brand-new name, they'll have to exit the program and start over. In Chapter 9, you'll use a menu system that lets users completely reset what they're doing without exiting.

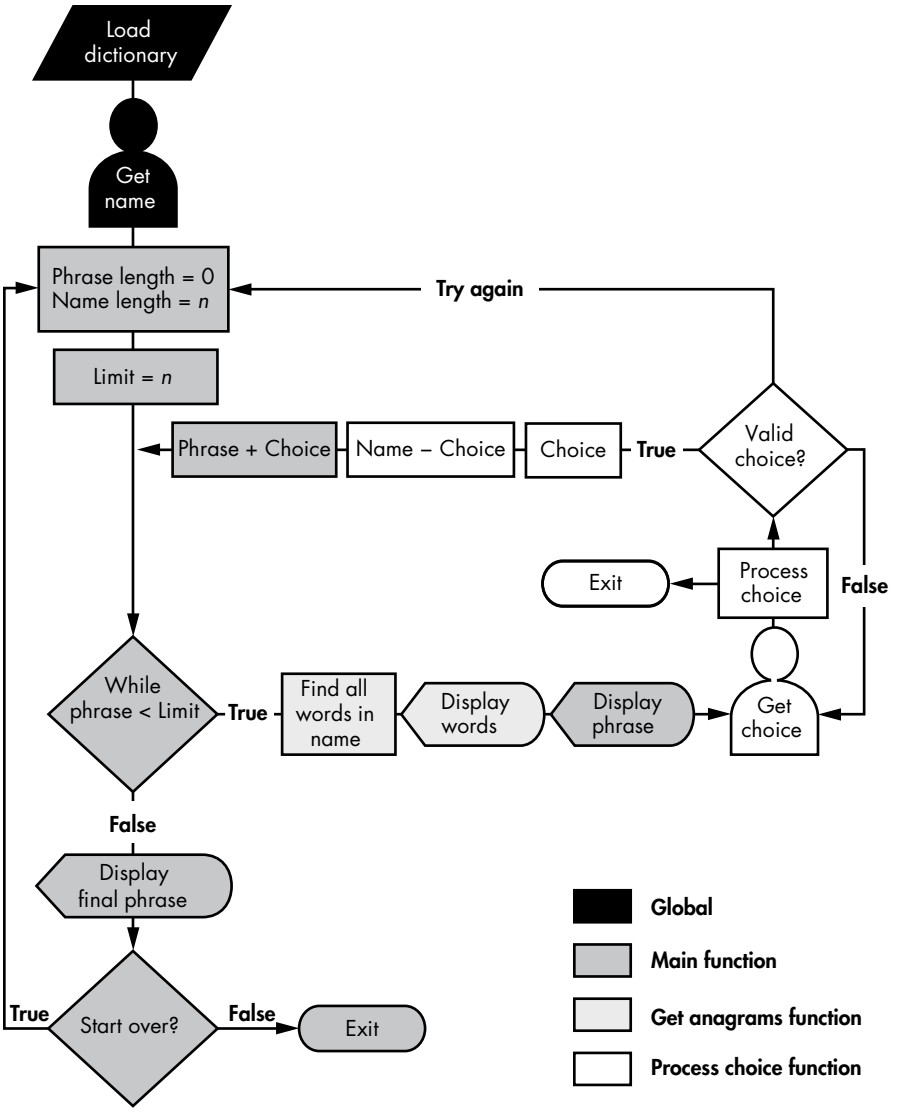


Figure 3-1: Flowchart for finding phrase anagrams with function assignments highlighted

The Anagram Phrase Code

The code in this section takes a name from a user and helps them build an anagram phrase of that name. You can download the entire script from *https://www.nostarch.com/impracticalpython/* as *phrase_anagrams.py*. You'll also need to download the *load_dictionary.py* program. Save both files in the same folder. You can use the same dictionary file you used in "Project #4: Finding Single-Word Anagrams" on page 36.

Setting Up and Finding Anagrams

Listing 3-2 imports the modules that *phrase_anagrams.py* uses, loads a dictionary file, asks the user for an input name, and defines the `find_anagrams()` function, which does most of the work related to finding anagrams.

phrase_anagrams.py, part 1

```
❶ import sys
  from collections import Counter
  import load_dictionary

❷ dict_file = load_dictionary.load('2of4brif.txt')
  # ensure "a" & "I" (both lowercase) are included
  dict_file.append('a')
  dict_file.append('i')
  dict_file = sorted(dict_file)

❸ ini_name = input("Enter a name: ")

❹ def find_anagrams(name, word_list):
      """Read name & dictionary file & display all anagrams IN name."""
   ❺ name_letter_map = Counter(name)
     anagrams = []
   ❻ for word in word_list:
       ❼ test = ''
       ❽ word_letter_map = Counter(word.lower())
       ❾ for letter in word:
             if word_letter_map[letter] <= name_letter_map[letter]:
                 test += letter
         if Counter(test) == word_letter_map:
             anagrams.append(word)
  ❿ print(*anagrams, sep='\n')
     print()
     print("Remaining letters = {}".format(name))
     print("Number of remaining letters = {}".format(len(name)))
     print("Number of remaining (real word) anagrams = {}".format(len(anagrams)))
```

Listing 3-2: Imports modules, loads dictionary, and defines the `find_anagrams()` function

Start with the `import` statements ❶, using the recommended order of Python Standard Library, third-party modules, and then locally developed modules. You need `sys` for coloring specific outputs red in the IDLE window and for letting the user exit the program with a keystroke. You'll use `Counter` to help identify anagrams of the input name.

Next load the dictionary file using the imported module ❷. The filename argument should be the filename of the dictionary you're using. Because some dictionary files omit *a* and *I*, append these (if needed), and sort the list so that they can be found at the proper alphabetical locations, rather than at the end of the list.

Now get a name from the user and assign it to the variable `ini_name` (or "initial name") ❸. You'll derive a `name` variable from this initial name, and

name will be progressively changed as the user builds up the name anagram. Preserving the initial name as a separate variable will let you reset everything if the user wants to start over or try again.

The next block of code is find_anagrams() ❹, the function for finding anagrams in the name. The parameters for this function consist of a name and a word list. The function starts by using Counter to count the number of times a given letter appears in the name and then assigns the count to the variable name_letter_map ❺; Counter uses a dictionary structure with the letter as the key and the count as the value. The function then creates an empty list to hold the anagrams and starts a for loop through each word in the dictionary file ❻.

The for loop starts by creating an empty string called test ❼. Use this variable to accumulate all the letters in the word that "fit" in name. Then make a Counter for the current word, as you did for name, and call it word_letter_map ❽. Loop through the letters in word ❾, checking that the count of each letter is the same as, or less than, the count in name. If the letter meets this condition, then it is added to the test string. Since some letters might get rejected, end the loop by running Counter on test and comparing it to word_letter_map. If they match, append the word to the anagrams list.

The function ends by displaying the list of words, using the splat operator with print, along with some statistics for the user ❿. Note that find_anagrams() doesn't return anything. This is where the human interaction part comes in. The program will continue to run, but nothing will happen until the user chooses a word from the displayed list.

Processing the User's Choice

Listing 3-3 defines process_choice(), the function in *phrase_anagrams.py* that takes the user's choice of word (or words), checks it against the remaining letters in the name variable, and returns acceptable choices—along with any leftover letters—to the main() function. Like main(), this function gets to talk directly to the user.

phrase_anagrams.py, part 2

```
❶ def process_choice(name):
    """Check user choice for validity, return choice & leftover letters."""
    while True:
      ❷ choice = input('\nMake a choice else Enter to start over or # to end: ')
        if choice == '':
            main()
        elif choice == '#':
            sys.exit()
        else:
          ❸ candidate = ''.join(choice.lower().split())
      ❹ left_over_list = list(name)
      ❺ for letter in candidate:
            if letter in left_over_list:
                left_over_list.remove(letter)
      ❻ if len(name) - len(left_over_list) == len(candidate):
            break
```

```
        else:
            print("Won't work! Make another choice!", file=sys.stderr)
❼ name = ''.join(left_over_list)  # makes display more readable
❽ return choice, name
```

Listing 3-3: Defines the process_choice() function

Start by defining the function with one parameter called `name` ❶. The first time the program is run, this parameter will be the same as the `ini_name` variable—the full name entered by the user when the program starts up. After the user has chosen a word (or words) to use in the anagram phrase, it will represent the remaining letters in the name.

Start the function with a `while` loop that will run until the user makes a valid choice and then get input from the user ❷. The user has a choice of entering one or more words from the current anagram list, pressing ENTER to start over, or pressing # to quit. Use #, rather than a word or letter, so that it can't be confused for a valid choice.

If the user makes a choice, the string is assigned to the variable `candidate`, stripped of whitespace, and converted to all lowercase ❸. This is so it can be directly compared to the `name` variable. After that, a list is built from the `name` variable to hold any remaining letters ❹.

Now begin a loop to subtract the letters used in `candidate` ❺. If a chosen letter is present in the list, it's removed.

If the user entered a word that isn't in the displayed list, or entered multiple words, a letter may not be present in the list. To check for this, subtract the leftover letters from `name` and, if the result is the number of letters in `candidate`, determine that the input is valid and break out of the `while` loop ❻. Otherwise, display a warning and color it red for those using the IDLE window. The `while` loop will keep prompting the user until an acceptable choice is made.

If all the letters in the user's choice pass the test, the list of leftovers is converted back into a string and used to update the `name` variable ❼. Converting the list into a string isn't strictly necessary, but it keeps the `name` variable type consistent and lets you display the remaining letters in a clearly readable format without the need for additional `print` arguments.

Finish by returning both the user's choice and the string of remaining letters (`name`) to the `main()` function ❽.

Defining the main() Function

Listing 3-4 defines the `main()` function in *phrase_anagrams.py*. This function wraps the previous functions, runs a `while` loop, and determines when the user has successfully created an anagram phrase.

phrase_anagrams.py, part 3

```
def main():
    """Help user build anagram phrase from their name."""
❶ name = ''.join(ini_name.lower().split())
   name = name.replace('-', '')
```

```
    ❷ limit = len(name)
      phrase = ''
      running = True

    ❸ while running:
        ❹ temp_phrase = phrase.replace(' ', '')
        ❺ if len(temp_phrase) < limit:
              print("Length of anagram phrase = {}".format(len(temp_phrase)))

            ❻ find_anagrams(name, dict_file)
              print("Current anagram phrase =", end=" ")
              print(phrase, file=sys.stderr)

            ❼ choice, name = process_choice(name)
              phrase += choice + ' '

        ❽ elif len(temp_phrase) == limit:
              print("\n*****FINISHED!!!*****\n")
              print("Anagram of name =", end=" ")
              print(phrase, file=sys.stderr)
              print()
            ❾ try_again = input('\n\nTry again? (Press Enter else "n" to quit)\n ')
              if try_again.lower() == "n":
                  running = False
                  sys.exit()
              else:
                  main()

❿ if __name__ == '__main__':
      main()
```

Listing 3-4: Defines and calls `main()` function

The first order of business is to turn the `ini_name` variable into a continuous string of lowercase characters with no whitespace ❶. Remember, case matters to Python, so convert all strings to lowercase wherever they occur; that way, comparisons will work as intended. Python also recognizes spaces as characters, so you need to remove these, as well as hyphens in hyphenated names, before doing any letter counts. By declaring this new `name` variable, you preserve the initial name in case the user wants to start over. Only `name` will be altered in the `process_choice()` function.

Next, get the length of the name ❷ to use as a limit in the `while` loop. This will let you know when the anagram phrase has used all the letters in the name and it's time to end the loop. Do this outside the `while` loop to ensure you are using the full initial name. Then assign a variable to hold the anagram phrase and set a `running` variable to `True` to control the `while` loop.

Now begins the big loop that lets you iterate over the name and build an anagram phrase ❸. First, prepare a string to hold the growing phrase and strip it of whitespace ❹. Spaces will count as letters and throw off the operator when the length of the phrase is compared to the `limit` variable.

Next, make the comparison, and if the length of the phrase is less than the limit, display the current length of the phrase as a prelude to engaging with the user ❺.

It's time to put the other functions to work. Call `find_anagrams()` ❻ and pass it the name and dictionary file to get the list of anagrams in the name. At the bottom of the displayed list, show the user the current phrase. Use the `print()` function's `end` parameter to display two `print` statements on the same line. This way, you can use a red font on the phrase in the IDLE window to distinguish it from all the other information in the display.

Next, call the `process_choice()` function ❼ to get the user's word choice and add it to the growing anagram phrase. This also gets the updated version of the `name` variable so that the program can use it again in the `while` loop in the event that the phrase isn't complete.

If the length of the phrase is equal to the `limit` variable ❽, the name anagram is complete. Let the user know they're finished and present the phrase using red font. Note that you don't have a conditional for the length of the phrase being greater than the `limit` variable. That's because the `process_choice()` function is already handling this outcome (choosing more letters than are available would not pass the validation criterion).

The `main()` function ends by asking the user whether they want to try again. If they type `n`, the program ends; if they press ENTER, the `main()` function is called again ❾. As stated earlier, the only way for the user to change the initial name is to exit and relaunch the program.

Outside of the `main()` function, end with the standard two lines for calling the `main()` function when the program is not imported as a module ❿.

Running an Example Session

In this section, I've included an example interactive session, using *phrase_anagrams.py* and the name *Bill Bo*. **`Bold`** font indicates user input, and ***`italic bold`*** font indicates where red font is used in the display.

```
Enter a name: Bill Bo
Length of anagram phrase = 0
bib
bill
blob
bob
boil
boll
i
ill
lib
lilo
lo
lob
oi
oil
```

```
Remaining letters = billbo
Number of remaining letters = 6
Number of remaining (real word)anagrams = 14
Current anagram phrase =

Make a choice else Enter to start over or # to end: ill
Length of anagram phrase = 3
bob

Remaining letters = bbo
Number of remaining letters = 3
Number of remaining (real word)anagrams = 1
Current anagram phrase = ill

Make a choice else Enter to start over or # to end: Bob

***** FINISHED!!! *****

Anagram of name = ill Bob


Try again? (Press Enter else "n" to quit)
```

The number of anagrams found depends on the dictionary file you use. If you're having a hard time building anagram phrases, try using a larger dictionary.

Project #6: Finding Voldemort: The Gallic Gambit

Did you ever wonder how Tom Riddle came up with the anagram "I am Lord Voldemort"? Did he put quill to parchment or just wave a wand? Could the magic of Python have helped?

Let's pretend for a moment that you're the professor of computer wizardry at Hogwarts, and Tom Riddle, school prefect and model student, has come to you for help. Using your *phrase_anagrams.py* spell from the previous section, he could find *I am Lord* in the very first list of anagrams, much to his delight. But the remaining letters, *tmvoordle*, yield only trivial words like *dolt*, *drool*, *looter*, and *lover*. Riddle would not be pleased.

In hindsight, the problem is apparent: *Voldemort* is French and won't be found in any English dictionary file. *Vol de la mort* means "flight of death" in French, so Voldemort is loosely "death flight." But Riddle is 100 percent English, and so far, you have been working with English. Without reverse engineering, you have no more reason to suddenly switch out your English dictionary for a French one than you have to use Dutch, German, Italian, or Spanish.

You *could* try randomly shuffling the remaining letters and seeing what falls out. Unfortunately, the number of possible combinations is the factorial of the number of letters divided by the factorial of the number of repeats (*o* occurs twice): 9! / 2! = 181,440. If you were to scroll through all

those permutations, taking only one second to review each, it would take you over two days to complete the list! And if you asked Tom Riddle to do this, he would probably use you to make a horcrux!

At this point, I would like to explore two logical paths ahead. One I call the "Gallic Gambit" and the other the "British Brute-Force." We'll look at the first one here and the second one in the next section.

NOTE Marvolo *is clearly a* fabricated word *used to make the Voldemort anagram work. J.K. Rowling could have gained additional latitude by using* Thomas *for* Tom *or by leaving off the* Lord *or* I am *parts. Tricks like these are used when the book is translated into non-English languages. In some languages, one or both names may need to be changed. In French, the anagram is "I am Voldemort." In Norwegian, "Voldemort the Great." In Dutch, "My name is Voldemort." In others, like Chinese, the anagram can't be used at all!*

Tom Riddle was obsessed with beating death, and if you go looking for death in *tmvoordle*, you will find both the old French *morte* (as in the famous book *Le Morte d'Arthur* by Sir Thomas Malory) and the modern French *mort*. Removing *mort* leaves *vodle*, five letters with a very manageable number of permutations. In fact, you can easily find *volde* right in the interpreter window:

```
❶ >>> from itertools import permutations
  >>> name = 'vodle'
❷ >>> perms = [''.join(i) for i in permutations(name)]
❸ >>> print(len(perms))
  120
❹ >>> print(perms)
  ['vodle', 'vodel', 'volde', 'voled', 'voedl', 'voeld', 'vdole', 'vdoel',
  'vdloe', 'vdleo', 'vdeol', 'vdelo', 'vlode', 'vloed', 'vldoe', 'vldeo',
  'vleod', 'vledo', 'veodl', 'veold', 'vedol', 'vedlo', 'velod', 'veldo',
  'ovdle', 'ovdel', 'ovlde', 'ovled', 'ovedl', 'oveld', 'odvle', 'odvel',
  'odlve', 'odlev', 'odevl', 'odelv', 'olvde', 'olved', 'oldve', 'oldev',
  'olevd', 'oledv', 'oevdl', 'oevld', 'oedvl', 'oedlv', 'oelvd', 'oeldv',
  'dvole', 'dvoel', 'dvloe', 'dvleo', 'dveol', 'dvelo', 'dovle', 'dovel',
  'dolve', 'dolev', 'doevl', 'doelv', 'dlvoe', 'dlveo', 'dlove', 'dloev',
  'dlevo', 'dleov', 'devol', 'devlo', 'deovl', 'deolv', 'delvo', 'delov',
  'lvode', 'lvoed', 'lvdoe', 'lvdeo', 'lveod', 'lvedo', 'lovde', 'loved',
  'lodve', 'lodev', 'loevd', 'loedv', 'ldvoe', 'ldveo', 'ldove', 'ldoev',
  'ldevo', 'ldeov', 'levod', 'levdo', 'leovd', 'leodv', 'ledvo', 'ledov',
  'evodl', 'evold', 'evdol', 'evdlo', 'evlod', 'evldo', 'eovdl', 'eovld',
  'eodvl', 'eodlv', 'eolvd', 'eoldv', 'edvol', 'edvlo', 'edovl', 'edolv',
  'edlvo', 'edlov', 'elvod', 'elvdo', 'elovd', 'elodv', 'eldvo', 'eldov']
  >>>
❺ >>> print(*perms, sep='\n')
  vodle
  vodel
  volde
  voled
  voedl
  --snip--
```

Start by importing `permutations` from `itertools` ❶. The `itertools` module is a group of functions in the Python Standard Library that create iterators for efficient looping. You generally think of permutations of *numbers*, but the `itertools` version works on *elements* in an iterable, which includes letters.

After entering the name or, in this case, the remaining letters in the name, use list comprehension to create a list of permutations of the name ❷. Join each element in a permutation so each item in the final list will be a unique permutation of *vodle*. Using `join` yields the new name as an element, `'vodle'`, versus a hard-to-read tuple of single-character elements,`('v', 'o', 'd', 'l', 'e')`.

Get the length of the permutations as a check; that way, you can confirm that it is, indeed, the factorial of 5 ❸. At the end, no matter how you print it ❹❺, *volde* is easy to find.

Project #7: Finding Voldemort: The British Brute-Force

Now let's assume Tom Riddle is bad at anagrams (or French). He doesn't recognize *mort* or *morte*, and you're back to shuffling the remaining nine letters thousands and thousands of times, looking for a combination of letters that he would find pleasing.

On the bright side, this is a more interesting problem programmatically than the interactive solution you just saw. You just need to whittle down all the permutations using some form of filtering.

THE OBJECTIVE

Reduce the number of anagrams of *tmvoordle* to a manageable number that will still contain *Voldemort*.

Strategy

Per the *Oxford English Dictionary, 2nd Edition*, there are 171,476 English words currently in use, which is fewer than the total number of permutations in *tmvoordle*! Regardless of the language, you can surmise that most of the anagrams generated by the `permutations()` function are nonsense.

With *cryptography*, the science of codes and ciphers, you can safely eliminate many useless, unpronounceable combinations, such as *ldtmvroeo*, and you won't even have to inspect them visually. Cryptographers have long studied languages and compiled statistics on recurring patterns of words and letters. We can use many cryptanalytical techniques for this project, but let's focus on three: consonant-vowel mapping, trigram frequency, and digram frequency.

Filtering with Consonant-Vowel Mapping

A *consonant-vowel map* (*c-v map*) simply replaces the letters in a word with a *c* or a *v*, as appropriate. *Riddle*, for example, becomes *cvcccv*. You can write a program that goes through a dictionary file and creates c-v maps for each word. By default, impossible combinations, like *ccccccvvv*, will be excluded. You can further exclude membership by removing words with c-v maps that are *possible* but that have a low frequency of occurrence.

C-v maps are fairly inclusive, but that's good. An option for *Riddle* at this point is to make up a new proper name, and proper names don't have to be words that occur in a dictionary. So you don't want to be *too* exclusive early in the process.

Filtering with Trigrams

Since the initial filter needs a relatively wide aperture, you'll need to filter again at a lower level to safely remove more anagrams from the permutations. *Trigrams* are triplets comprising three consecutive letters. It should come as no surprise that the most common trigram in English is the word *the*, followed closely by *and* and *ing*. At the other end of the scale are trigrams like *zvq*.

You can find statistics on the frequency of occurrence of trigrams online at sites like *http://norvig.com/ngrams/count_3l.txt*. For any group of letters, like *tmvoordle*, you can generate and use a list of the least common trigrams to further reduce the number of permutations. For this project, you can use the *least-likely_trigrams.txt* file, downloadable from *https://www.nostarch.com/impracticalpython/*. This text file contains the trigrams in *tmvoordle* that occur in the bottom 10 percent of trigrams in the English language, based on frequency of occurrence.

Filtering with Digrams

Digrams (also called *bigrams*) are letter pairs. Commonly occurring digrams in English include *an*, *st*, and *er*. On the other hand, you rarely see pairs like *kg*, *vl*, or *oq*. You can find statistics on the frequency of occurrence of digrams at websites such as *https://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/digraphs.html* and *http://practicalcryptography.com/*.

Table 3-1 was built from the *tmvoordle* collection of letters and a 60,000-word English dictionary file. The letters along the left side of the chart are the starting letters for the digrams; those along the top represent the end letter. For example, to find *vo*, start with the *v* on the left and read across to the column beneath the *o*. For the digrams found in *tmvoordle*, *vo* occurs only 0.8 percent of the time.

Table 3-1: Relative Frequency of Digrams from the Letters *tmvoordle* in a 60,000-Word Dictionary (Black Squares Indicate No Occurrences)

	d	e	l	m	o	r	t	v
d		3.5%	0.5%	0.1%	1.7%	0.5%	0.0%	0.1%
e	6.6%		2.3%	1.4%	0.7%	8.9%	2.0%	0.6%
l	0.4%	4.4%		0.1%	4.2%	0.0%	0.4%	0.1%
m	0.0%	2.2%	0.0%		2.8%	0.0%	0.0%	0.0%
o	1.5%	0.5%	3.7%	3.2%	5.3%	7.1%	2.4%	1.4%
r	0.9%	6.0%	0.4%	0.7%	5.7%		1.3%	0.3%
t	0.0%	6.2%	0.6%	0.1%	3.6%	2.3%		0.0%
v	0.0%	2.5%	0.0%	0.0%	0.8%	0.0%	0.0%	

Assuming you're looking for "English-like" letter combinations, you can use frequency maps like this to exclude letter pairs that are unlikely to occur. Think of it as a "digram sieve" that lets only the unshaded squares pass.

To be safe, just exclude digrams that occur less than 0.1 percent of the time. I've shaded these in black. Notice that it would be very easy to eliminate the required *vo* pairing in *Voldemort*, if you cut too close to the bone!

You can design your filter to be even more selective by tagging digrams that are unlikely to occur at the start of a word. For example, while it's not unusual for the digram *lm* to occur *within* a word (as in *almanac* and *balmy*), you'll need a lot of luck finding a word that *starts* with *lm*. You don't need cryptography to find these digrams; just try to pronounce them! Some starting-point choices for these are shaded gray in Table 3-2.

Table 3-2: Update of Table 3-1, Where Gray-Shaded Squares Indicate Digrams Unlikely to Occur at the Start of a Word

	d	e	l	m	o	r	t	v
d		3.5%	0.5%	0.1%	1.7%	0.5%	0.0%	0.1%
e	6.6%		2.3%	1.4%	0.7%	8.9%	2.0%	0.6%
l	0.4%	4.4%		0.1%	4.2%	0.0%	0.4%	0.1%
m	0.0%	2.2%	0.0%		2.8%	0.0%	0.0%	0.0%
o	1.5%	0.5%	3.7%	3.2%	5.3%	7.1%	2.4%	1.4%
r	0.9%	6.0%	0.4%	0.7%	5.7%		1.3%	0.3%
t	0.0%	6.2%	0.6%	0.1%	3.6%	2.3%		0.0%
v	0.0%	2.5%	0.0%	0.0%	0.8%	0.0%	0.0%	

You now have three filters you can use on the 181,440 permutations of *tmvoordle*: c-v maps, trigrams, and digrams. As a final filter, you should give the user the option of viewing only anagrams that start with a given letter. This will let the user divide the remaining anagrams into more manageable "chunks," or focus on the more intimidating-sounding anagrams, like those that begin with *v*!

The British Brute-Force Code

The upcoming code generates permutations of *tmvoordle* and passes them through the filters just described. It then gives the user the option to view either all the permutations or only those starting with a given letter.

You can download all the programs you'll need from *https://www.nostarch.com/impracticalpython/*. The code in this section is one script named *voldemort_british.py*. You'll also need the *load_dictionary.py* program in the same folder, along with the same dictionary file you used for the projects earlier in this chapter. Finally, you'll need a new file named *least-likely_trigrams.txt*, a text file of trigrams with a low frequency of occurrence in English. Download all these files into the same folder.

Defining the main() Function

Listing 3-5 imports the modules that *voldemort_british.py* needs and defines its `main()` function. In the *phrase_anagrams.py* program, you defined the `main()` function at the end of the code. Here we put it at the start. The advantage is that you can see what the function is doing—how it's running the program—from the start. The disadvantage is that you don't know what any of the helper functions do yet.

voldemort_british.py, part 1

```
❶ import sys
  from itertools import permutations
  from collections import Counter
  import load_dictionary

❷ def main():
      """Load files, run filters, allow user to view anagrams by 1st letter."""
   ❸ name = 'tmvoordle'
     name = name.lower()

   ❹ word_list_ini = load_dictionary.load('2of4brif.txt')
     trigrams_filtered = load_dictionary.load('least-likely_trigrams.txt')

   ❺ word_list = prep_words(name, word_list_ini)
     filtered_cv_map = cv_map_words(word_list)
     filter_1 = cv_map_filter(name, filtered_cv_map)
     filter_2 = trigram_filter(filter_1, trigrams_filtered)
     filter_3 = letter_pair_filter(filter_2)
     view_by_letter(name, filter_3)
```

Listing 3-5: Imports modules and defines the `main()` function

Start by importing modules you've used in the previous projects ❶. Now define the `main()` function ❷. The `name` variable is a string of the remaining letters *tmvoordle* ❸. Set it to lowercase to guard against a user input error. Next, use the `load_dictionary` module to load your dictionary file and the trigrams file as lists ❹. Your dictionary filename may be different from that shown.

Finally, call all the various functions in order ❺. I'll describe each of these functions momentarily, but basically, you need to prepare the word list, prepare the c-v maps, apply the three filters, and let the user view all the anagrams at once or view a subset based on the anagram's first letter.

Preparing the Word List

Listing 3-6 prepares the word list by including just the words that have as many letters as in the `name` variable (in this case, nine). You should also ensure that all the words are lowercase, to be consistent.

voldemort_british.py, part 2

```
❶ def prep_words(name, word_list_ini):
      """Prep word list for finding anagrams."""
   ❷ print("length initial word_list = {}".format(len(word_list_ini)))
      len_name = len(name)
   ❸ word_list = [word.lower() for word in word_list_ini
                  if len(word) == len_name]
   ❹ print("length of new word_list = {}".format(len(word_list)))
   ❺ return word_list
```

Listing 3-6: Creates lists of words that are equal in length to the `name` variable

Define the `prep_words()` function to take a name string and list of dictionary words as arguments ❶. I suggest that you print the lengths of your various word lists before and after they've gone through a filter; that way, you can track how much impact the filters are having. So print the length of the dictionary ❷. Assign a variable to hold the length of the name and then use list comprehension to create a new list by looping through the words in `word_list_ini`, keeping those whose length is the same as the number of letters in `name`, and converting them to lowercase ❸. Next, print the length of this new word list ❹, and finally, return this new list for use in the next function ❺.

Generating the C-V Map

You need to convert the prepared word list to a c-v map. Remember that you're no longer interested in actual words in the dictionary; those have been reviewed and rejected. Your goal is to shuffle the remaining letters until they form something that resembles a proper noun.

Listing 3-7 defines a function that generates c-v maps for each word in `word_list`. The program, *voldemort_british.py*, will use the c-v map to judge whether a shuffled letter combination is reasonable based on consonant-vowel patterns in the English language.

voldemort_british.py, part 3

```
❶ def cv_map_words(word_list):
      """Map letters in words to consonants & vowels."""
   ❷ vowels = 'aeiouy'
   ❸ cv_mapped_words = []
   ❹ for word in word_list:
         temp = ''
         for letter in word:
             if letter in vowels:
                 temp += 'v'
             else:
                 temp += 'c'
         cv_mapped_words.append(temp)

     # determine number of UNIQUE c-v patterns
   ❺ total = len(set(cv_mapped_words))
     # target fraction to eliminate
   ❻ target = 0.05
     # get number of items in target fraction
   ❼ n = int(total * target)
   ❽ count_pruned = Counter(cv_mapped_words).most_common(total - n)
   ❾ filtered_cv_map = set()
     for pattern, count in count_pruned:
         filtered_cv_map.add(pattern)
     print("length filtered_cv_map = {}".format(len(filtered_cv_map)))
   ❿ return filtered_cv_map
```

Listing 3-7: Generates c-v maps from the words in `word_list`

Define the `cv_map_words()` function to take the prepped word list as an argument ❶. Since consonants and vowels form a binary system, you can define the vowels with a string ❷. Create an empty list to hold the maps ❸. Then loop through the words and the letters in each word, converting the letters to either a *c* or *v* ❹. Use a variable called `temp` to accumulate the map; then append it to the list. Note that `temp` is reinitialized each time the loop repeats.

You want to know the frequency of occurrence of a given c-v map pattern (for example, *cvcv*), so you can remove those with a low likelihood of occurrence. Before calculating the frequency, you need to collapse your list down to unique c-v maps—as it is now, *cvcv* may be repeated many, many times. So, turn the `cv_mapped_words` list into a set, to remove duplicates, and get its length ❺. Now you can define a target percentage to eliminate, using fractional values ❻. Start with a low number like `0.05`—equivalent to 5 percent—so you're less likely to eliminate anagrams that can form usable proper names. Multiply this target value by the total length of the `cv_mapped_words` set and assign the result to the variable `n` ❼. Be sure to convert `n` to an integer; since it will represent a count value, it can't be a float.

The `Counter` module data type has a handy method, `most_common()`, that will return the most common items in a list based on a *count* value that you provide; in this case, that value will be the length of the c-v map list, `total`, minus `n`. The value you pass `most_common()` must be an integer. If you pass

the `most_common()` function the length of the list, it will return all the items in the list. If you subtract the count for the least likely 5 percent, you will effectively eliminate these c-v maps from the list ❽.

Remember, `Counter` returns a dictionary, but all you need are the final c-v maps, not their associated frequency counts. So initialize an empty set called `filtered-cv-map` ❾ and loop through each key-value pair in `count_pruned()`, adding only the key to the new set. Print the length of this set, so you can see the impact of the filter. Then finish by returning the filtered c-v map for use in the next function ❿.

Defining the C-V Map Filter

Listing 3-8 applies the c-v map filter: anagrams are generated based on permutations of the letters in the `name` variable, and then the program converts them to c-v maps and compares those anagrams to the filtered c-v maps built with the `cv_map_words()` function. If an anagram's c-v map is found in `filtered_cv_map`, then the program stores the anagram for the next filter.

voldemort_british.py, part 4

```
❶ def cv_map_filter(name, filtered_cv_map):
    """Remove permutations of words based on unlikely cons-vowel combos."""
  ❷ perms = {''.join(i) for i in permutations(name)}
    print("length of initial permutations set = {}".format(len(perms)))
    vowels = 'aeiouy'
  ❸ filter_1 = set()
  ❹ for candidate in perms:
        temp = ''
        for letter in candidate:
            if letter in vowels:
                temp += 'v'
            else:
                temp += 'c'
      ❺ if temp in filtered_cv_map:
            filter_1.add(candidate)
    print("# choices after filter_1 = {}".format(len(filter_1)))
  ❻ return filter_1
```

Listing 3-8: Defines `cv_map_filter()` function

Define the function `cv_map_filter()` to take two arguments: the name, followed by the set of c-v maps returned by `cv_map_words()` ❶. Use set comprehension and the `permutations` module to generate the set of permutations ❷. I described this process in "Project #6: Finding Voldemort: The Gallic Gambit" on page 49. Use a set here to permit later use of set operations, like taking the difference between two filter sets. This also removes duplicates, as `permutations` treats each *o* as a separate item, and returns 9!, rather than 9! / 2!. Note that `permutations` considers *tmvo**o**rdle* and *tmvo**o**rdle* different strings.

Now initialize an empty set to hold the contents of the first filter ❸ and begin looping through the permutations ❹. Use the term *candidate*, as most of these aren't words but just strings of random letters. For each candidate,

loop through the letters and map them to a *c* or a *v*, as you did with the `cv_words()` function. Check each c-v map, `temp`, for membership in `filtered_cv_map`. This is one reason for using sets: membership checks are very fast. If the candidate meets the condition, add it to `filter_1` ❺. Finish by returning your new anagram set ❻.

Defining the Trigram Filter

Listing 3-9 defines the trigram filter, which removes the permutations with unlikely three-letter triplets. It uses a text file derived from various cryptography websites that has been tailored to the letters in *tmvoordle*. This function will return only permutations that include one of these trigrams; the `main()` function will pass the new set to the next filter function.

voldemort_british.py, part 5

```
❶ def trigram_filter(filter_1, trigrams_filtered):
      """Remove unlikely trigrams from permutations."""
   ❷ filtered = set()
   ❸ for candidate in filter_1:
       ❹ for triplet in trigrams_filtered:
             triplet = triplet.lower()
             if triplet in candidate:
                 filtered.add(candidate)
   ❺ filter_2 = filter_1 - filtered
     print("# of choices after filter_2 = {}".format(len(filter_2)))
   ❻ return filter_2
```

Listing 3-9: Defines the `trigram_filter()` function

Parameters for the trigram filter include the output from the c-v map filter and the external list of unlikely trigrams, `trigrams_filtered` ❶.

Initialize an empty set to hold permutations that contain one of the forbidden trigrams ❷. Then start another `for` loop that looks through the candidates that survived the last filter ❸. A nested `for` loop looks at each triplet in the trigrams list ❹. If the triplet is in the candidate, it is added to the filter.

Now you can use set operations to subtract the new filter from `filter_1` ❺ and then return the difference for use with the next filter ❻.

Defining the Digram Filter

Listing 3-10 defines the digram filter, which removes unlikely letter pairs. Some will trigger the filter if they occur anywhere within the permutation; others will do so only if they occur at the start of the permutation. The disallowed digrams are based on the shaded cells in Table 3-2. The function returns the results of this filter for use in the final filter function.

voldemort_british.py, part 6

```
❶ def letter_pair_filter(filter_2):
      """Remove unlikely letter-pairs from permutations."""
   ❷ filtered = set()
   ❸ rejects = ['dt', 'lr', 'md', 'ml', 'mr', 'mt', 'mv',
                'td', 'tv', 'vd', 'vl', 'vm', 'vr', 'vt']
```

```
❹ first_pair_rejects = ['ld', 'lm', 'lt', 'lv', 'rd',
                       'rl', 'rm', 'rt', 'rv', 'tl', 'tm']
❺ for candidate in filter_2:
    ❻ for r in rejects:
          if r in candidate:
              filtered.add(candidate)
    ❼ for fp in first_pair_rejects:
          if candidate.startswith(fp):
              filtered.add(candidate)
❽ filter_3 = filter_2 - filtered
  print("# of choices after filter_3 = {}".format(len(filter_3)))
❾ if 'voldemort' in filter_3:
      print("Voldemort found!", file=sys.stderr)
❿ return filter_3
```

Listing 3-10: Defines the `letter_pair_filter()` function

This filter accepts the results of the previous filter as an argument ❶. An empty set is initialized to hold any discarded permutations ❷. Then two lists of rejected pairs are assigned to the variables `rejects` ❸ and `first_pair_rejects` ❹. Both lists were entered manually. The first represents cells shaded black in Table 3-2; the second references cells shaded gray. Any permutation that contains a member of the first list—anywhere—will be discarded; permutations that *start with* a member of the second list will not be allowed. You can add or remove digrams to these lists to change how the filter behaves.

Begin looping through the permutations—continue to refer to these as "candidates," as they aren't necessarily words ❺. A nested `for` loop goes through the pairs in `rejects`, determines whether any are in `candidate`, and adds them to the `filtered` set ❻. A second nested `for` loop repeats this process for the `first_pair_rejects` ❼. Subtract `filtered` from the set returned from the previous function, `filter_2` ❽.

For fun *and* to ensure you haven't filtered too far, check whether *voldemort* is included in `filter_3` ❾ and print an announcement to highlight the discovery, using eye-catching red font for IDLE users. Then finish by returning the final filtered set ❿.

Letting the User Choose the Starting Letter

You don't know ahead of time whether your filtering will be successful. You may still end up with thousands of permutations. Providing the option to look at only a subset of the output won't reduce the overall number, but it will make it *psychologically* easier to face. Listing 3-11 adds, to *voldemort_british.py*, the ability to view a list of anagrams that begin with a certain input letter.

voldemort_british.py, part 7

```
❶ def view_by_letter(name, filter_3):
      """Filter to anagrams starting with input letter."""
   ❷ print("Remaining letters = {}".format(name))
   ❸ first = input("select a starting letter or press Enter to see all: ")
   ❹ subset = []
```

```
❺ for candidate in filter_3:
      if candidate.startswith(first):
          subset.append(candidate)
❻ print(*sorted(subset), sep='\n')
  print("Number of choices starting with {} = {}".format(first, len(subset)))
❼ try_again = input("Try again? (Press Enter else any other key to Exit):")
  if try_again.lower() == '':
   ❽ view_by_letter(name, filter_3)
  else:
   ❾ sys.exit()
```

Listing 3-11: Defines the `view_by_letter()` function

Define the `view_by_letter()` function to take both the `name` variable and `filter_3` as arguments ❶. You need the name so you can show the user the available letter choices on which to filter ❷. Get the user's input on whether they want to see all the remaining permutations or just those beginning with a certain letter ❸. Then start an empty list to hold the latter subset ❹.

A `for` loop, with a conditional, checks whether a candidate starts with the chosen letter and appends those letters that pass to `subset` ❺. This list is printed with the splat operator ❻. Then the program asks the user whether they want to try again or exit ❼. If they press ENTER, then `view_by_letter()` is called, recursively, and runs again from the start ❽. Otherwise, the program exits ❾. Note that Python has a default recursion depth limit of 1,000, which we'll ignore in this project.

Running the main() Function

Back in the global space, Listing 3-12 completes the code by calling the `main()` function if the user runs the program in stand-alone mode versus importing into another program.

voldemort_british.py, part 8

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

Listing 3-12: Calls the `main()` function

Example output from the completed program is shown below. After the program applies the third filter, there are 248 permutations remaining, of which a very manageable 73 start with *v*. I've omitted the printout of the permutations for brevity. As noted in the output, *voldemort* survives the filtering.

```
length initial word_list = 60388
length of new word_list = 8687
length filtered_cv_map = 234
length of initial permutations set = 181440
# choices after filter_1 = 123120
# of choices after filter_2 = 674
```

```
# of choices after filter_3 = 248
Voldemort found!
Remaining letters = tmvoordle
select a starting letter or Enter to see all: v
```

Interestingly, another surviving permutation is *lovedmort*. Given how many people Voldemort killed—or had killed—this may be the most appropriate moniker of all.

Summary

In this chapter, you first wrote code that found the anagrams for a given word or name. You then expanded on this to find phrasal name anagrams, working interactively with the user. Finally, you employed cryptanalytical techniques to tease *Voldemort* out of almost 200,000 possible anagrams. Along the way, you applied useful functionality in the `collections` and `itertools` modules.

Further Reading

The *Jumble* website is *http://www.jumble.com/*.

You can find some representative online anagram generators at the following sites:

- *http://wordsmith.org/anagram/*
- *https://www.dcode.fr/anagram-generator*
- *http://www.wordplays.com/anagrammer/*

More anagram programs are found in *Think Python, 2nd Edition* (O'Reilly, 2015) by Allen Downey.

Cracking Codes with Python (No Starch Press, 2017) by Al Sweigart provides more code for computing word patterns, such as those used for filtering in the *voldemort_british.py* program.

Practice Project: Finding Digrams

You *could* comb through cryptography websites looking for frequency statistics, or you could derive them for yourself. Write a Python program that finds all the digrams in *tmvoordle* and then counts their frequency of occurrence in a dictionary file. Be sure to test your code on words like *volvo*, so you don't overlook repeating digrams in the same word. You can find a solution in the appendix or download *count_digrams_practice.py* from *https://www.nostarch.com/impracticalpython/*.

Challenge Project: Automatic Anagram Generator

Look at the online anagram generators I just referenced in "Further Reading" and write a Python program that mimics one of these. Your program should automatically generate phrase anagrams from an input name and display a subset (for example, the first 500) for the user to review.