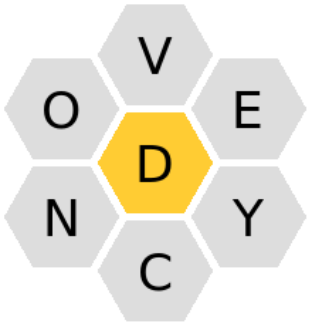
Project 2: Spelling Bee

The purpose of this project is to give you practice with strings, which are discussed in Chapter 6. In addition, implementing the graphical operations required for the later milestones will give you the chance to work with the `GCompound` and `GPolygon` classes introduced in Chapter 5. To help you with navigating this guide, there is a linked table of contents below, but you should ensure you read everything so that you don't miss out on important instructions.

Contents

The Spelling Bee Puzzle

One of the most popular features in the *New York Times* (and one that produces a surprisingly large revenue stream for the paper) is the Spelling Bee, which appears each day on the web at https://www.nytimes.com/puzzles/spelling-bee. Each Spelling Bee puzzle consists of seven hexagons (the bestagon) arranged in a small beehive-like shape. For example, the Spelling Bee puzzle from October 2, 2020, looked like:



Your task in the puzzle is to find as many words in this layout as you can, subject to the following rules:

- Each word must be at least four letters long, which means that the word `CON` is too short to be acceptable.
- Each word must not contain any letters other than the seven letters in the layout, although it *is legal to use the same letter more than once*. For example, you could form the word `COCOON` by using the `C` twice and the `O` three times.
- Each word must contain the center letter at least once, and so `CONE` would not be acceptable.
- Each word must be a valid English word. The *New York Times* uses a dictionary of "common" words that is more restrictive than a standard dictionary. Unfortunately, the *New York Times* does not publish a list of the words it considers legal, so your project will instead use the larger dictionary implemented by the `english` library module, which is included in the starter repository.

To get a sense for how the puzzle works, you should try to find the legal words in the puzzle shown above before looking at the solution in Figure 1.

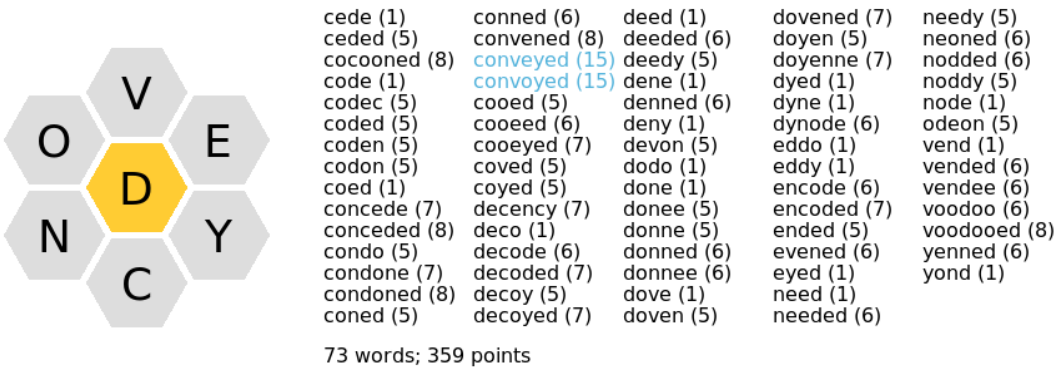


Figure 1: The solution to the Spelling Bee puzzle from October 2, 2020

Figure 1 shows an image of the final version of Project 2, assuming that the letters in the puzzle are the same ones shown on the previous page. In addition to the beehive-shaped diagram, the screen shows several columns listing the words one can find in this puzzle, the number of points assigned to each word, and a summary line at the bottom showing the total number of words and points. A four-letter word is worth just one point, but longer words score the number of letters they contain, so that a five-letter word is worth 5 points, a six-letter word is word six points, and so on. Words that use all seven letters in the puzzle are called **pangrams**, which score their length *plus* a bonus of seven points. This puzzle has two pangrams (there is always at least one) which are `conveyed` and `convoyed`. Both score 15 points (8 for the word plus an additional 7 for the pangram bonus).

The Starter Repository

As with Breakout, you don't have to implement the Spelling Bee project entirely from scratch. The repository to get you going can be found here, and includes the starter version of `SpellingBee.py`. Most of the starter file consists of definitions for the constants you need for the later milestones and a skeletal version of the program you need to write for Milestone 1.

In addition to the `SpellingBee.py` file, the repository also contains the latest version of the `pgl.py` graphics library, the `english.py` library that defines a list of valid English words, and the `DrawHexagon.py` program from the book, which includes a useful function to create a hexagon should you choose to use it.

Milestones

Your job in this project is to implement a solver for the Spelling Bee puzzle that lists all the words that can be formed from a particular seven-letter configuration, just like the display in Figure 1. You should not, however, try to get the entire project running all at once. Whenever you are faced with a large programming project, the most effective strategy is to define a series of milestones that allow you to complete the project in stages.

Ideally, each milestone you choose should be a program that you can test and debug independently, even if the code you write to test a particular milestone doesn't make its way into the final project. *The advantage you get from making it possible to test each stage more than compensates for having to write a little extra code along the way.* Similarly, it often makes sense to defer the more complex aspects of a project until after you have gotten the basic foundation working. The milestones described in this guide, for example, have you write a program that lists the words in a Spelling Bee puzzle on the console, before you try to display anything in the graphics window.

Milestone 1: List the solution on the console

Believe it or not, the first milestone–which is also the one that requires the least amount of code–does most of the work necessary for solving the Spelling Bee puzzle. All you have to do is write the implementation for the function:

```
def list_spelling_bee_words_on_console(puzzle):
```

This function takes a string of seven letters representing the Spelling Bee puzzle, starting with the letter in the center of the hexagon. The effect of the function is to list the words that can be formed from that set of letters, one per line. For Milestone 1, the starter file calls `list_spelling_bee_words_on_console` with the string `"DVONCYE"`, which corresponds to the puzzle shown in Figure 1. The advantage of starting with a particular puzzle is that you know exactly what the answers should look like. Given the dictionary in the `english` module, the output should look like Figure 2.

```
cede
ceded
cocooned
code
codec
coded
coden
codon
coed
concede
conceded
condo
condone
condoned
coned
conned
convened
conveyed
convoyed
cooed
cooeed
cooeyed
coved
coyed
decency
deco
decode
decoded
decoy
decoyed
deed
deeded
deedy
dene
denned
deny
devon
dodo
done
donee
donne
donned
donnee
dove
doven
dovened
doyen
doyenne
dyed
dyne
dynode
eddo
eddy
encode
encoded
ended
evened
eyed
need
needed
needy
neoned
nodded
noddy
node
odeon
vend
vended
vendee
voodoo
voodooed
yenned
yond
```

Figure 2: Example of the console printout of the valid words contained within the puzzle "DVONCYE"

This part of the project is much easier than it might at first appear. The `english` library exports a constant named `ENGLISH_WORDS`, which holds a list of all the English words. The code for `list_spelling_bee_words_on_console` (which you can copy into your code) therefore looks like this:

```
def list_spelling_bee_words_on_console(puzzle):
    """
    Displays the Spelling Bee words appearing in the puzzle.
    """
    puzzle = puzzle.lower()
    for word in ENGLISH_WORDS:
        if word_appears_in_puzzle(word, puzzle):
            print(word)
```

Your job is then to write a definition of the function `word_appears_in_puzzle` that checks whether the string passed as the parameter `word` is a legal solution for the Spelling Bee letters passed as the parameter `puzzle`. The conditions you need to check are:

- The word is at least 4 characters long.
- The word does not contain any letters other than the 7 letters in `puzzle`.
- The word contains the first letter in `puzzle`, which corresponds to the center of the hexagon.

Note that you do not need to check whether `word` is in the English dictionary because the structure of the program ensures that `word_appears_in_puzzle` is only called with valid dictionary words.

Milestone 2: Request a puzzle from the user

Once you have the `SpellingBee` program working with the `"DVONCYE"` example, your next task is to let the user enter the puzzle string so that you can verify that your program works with other Spelling Bee puzzles. Your program should use the build-in `input` function to read a string from the user, check to see that it contains seven letters with no duplication, and then call `list_spelling_bee_words_on_console` using that string as the puzzle. If the user enters a string that does not contain exactly seven unduplicated letters, your program should continually prompt the user to enter a new string until they enter a valid puzzle.

As an example, the user in the following sample run tries to enter a puzzle word that is too long and one with duplicated letters before entering the string `"XCINOPR"`:

```
Enter puzzle letters with center hex first: ABCDEFGH
That is not a legal puzzle. Please try again.
Enter puzzle letters with center hex first: ABCDEFA
That is not a legal puzzle. Please try again.
Enter puzzle letters with center hex first: XCINOPR
Words appearing in XCINOPR:
princox
```

The puzzle chosen on the user's third try happens to be the only combination of letters (at least in this dictionary) that generates no words besides a single seven-letter pangram. And, in case you are interested, the word *princox* is a 16^{th}-century term for a "pert youth" and appears in Shakespeare's *Romeo and Juliet* (Act I, scene 5).

Milestone 3: Add the scores to the console display

For this milestone, your job is to change the output format so that the program displays the score for each word and then prints a summary line at the end showing the number of words found and the total score. For example, the output for the puzzle `"LYCENTX"` (which appeared on January 8, 2000) should look like Figure 3.

```
Enter puzzle letters with center hex first: lycentx
Words appearing in lycentx:
cell (1)
celt (1)
cycle (5)
eely (1)
elect (5)
excel (5)
excellence (10)
excellency (10)
excellent (9)
excellently (18)
eyelet (6)
leet (1)
lent (1)
lenten (6)
lycee (5)
lynx (1)
nelly (5)
nettle (6)
nettly (6)
teel (1)
tele (1)
telex (5)
tell (1)
telly (5)
xylene (6)
xylyl (5)
yell (1)
27 words; 127 points
```

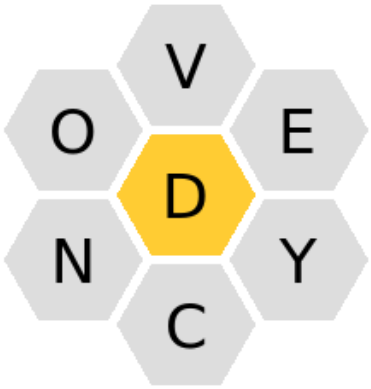
Figure 3: Example of showing both the individual word score next to each word, but also a summary line at the bottom with the total number of words and the total score available.

In the course of meeting this milestone, it can be useful to define functions for computing the score of a particular word and for checking to see if a particular word is a pangram (which effects its score). Ensure that you are outputting the correct scores for each word! The word `cell`, for instance, should only be worth 1 point while the word `excellently`, in

constrast, scores 18 points: 11 for the number of letters in the word and 7 for the pangram bonus.

Milestone 4: Displaying the Spelling Bee puzzle

Now that you have the string processing working, you are ready to start the graphical implementation. For Milestone 4, all you need to do is create the graphics window and display the letters in the beehive arrangement used in the *New York Times*. Given the puzzle `"DVONCYE"`, for example, your program should create the following diagram on the graphics window:



To ensure that you get some practice using the `GPolygon` and `GCompound` classes, you should implement this milestone by defining a function

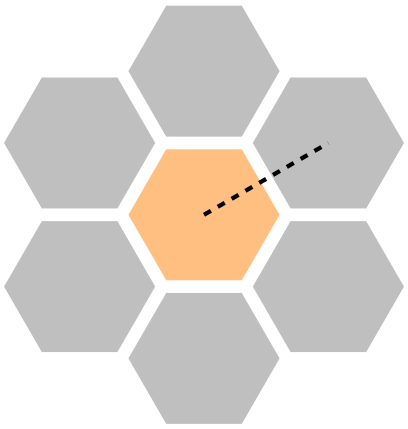
```
def create_beehive(puzzle):
```

which takes a string of seven letters representing the puzzle and returns a `GCompound` object that contains the `GObject` instances necessary to display the puzzle.

For each of the seven hexagons, there is a `GPolygon` object that creates the hexagon frame and a `GLabel` object that displays the letter in the center of the hexagon. The constants in the starter file specify all the details you need to determine the sizes, colors, fonts, and positions of these objects. The reference point for the `GCompound` object must be the center of the beehive, which is also the center of the interior hexagon. Thus, to display the example Spelling Bee puzzle at the correct location on the screen, all the main program has to do is execute the statements:

```
beehive = create_beehive("DVONCYE")
gw.add(beehive, BEEHIVE_X, BEEHIVE_Y)
```

Although most of the constants should be self-explanatory, there are two for which a little additional information may be helpful. The `HEX_LABEL_DY` constant indicates the distance from the center of the hexagon to the baseline of the label that shows the letter. The `HEX_SEP` constant specifies the distance between the centers of the hexagons in the beehive and therefore indicates the length of the dotted line in the following diagram:



This distance is the same for any of the outer hexagons. The only thing that changes is the angle at which the hexagon is drawn relative to the center point.

Although you can use trigonometry to calculate the positions of the outer hexagons, you can implement this part of the assignment much more easily than that. To generate any of the outer hexagons, all you have to do is create the hexagon as if it were in the center, and then use the `.move_polar()` method to shift it to the right position before adding it to the `GCompound`. The distance is always `HEX_SEP` and the angle varies from 30 to 330 degrees, increasing by 60 degrees each time.

Milestone 5: Display the words on the graphics window

The final milestone for this project requires you to integrate the code from Milestone 3 with the graphical display in Milestone 4 to produce an application that displays the word list on the graphics window as shown in Figure 1. Much of the code structure remains the same, but you have to figure out how to position the words on the graphics window. Calculating these positions is a bit tricky because the list of words is typically too long to display in a single column, which means that you need to write the code necessary to split the list across multiple columns. The left edge of the first column is given by the constant `WORDLIST_X`, and the baseline for the `GLabel` that display that word is given by the constant `WORDLIST_Y`. From there, words run down the first column, with each word positioned `WORDLIST_DY` pixels below the preceding one. When space in a column is exhausted, which means that the current baseline has gotten within `SCORE_BASELINE + SCORE_WORDLIST_SEP` pixels from the bottom of the window, your program should position the next word so that it is at the top of the next column, `WORDLIST_DX` pixels to the right of the preceding word. When you display the words in the list, your program should ordinarily set them in black using the font stored in the constant `WORDLIST_FONT`. If, however, the word is a pangram, it should use the color given by `PANGRAM_COLOR` instead.

Your last task in Milestone 5 is to add a line to the display that shows the number of words and the total score. This line should use `WORDLIST_X` as its x coordinate and `SCORE_BASELINE` pixels above the bottom of the window as its y coordinate.

Things to consider

- As with any large program, it is essential to get each milestone working before moving on to the next. It almost never works to write a large program all at once without testing the pieces as you go.
- You have to remember that uppercase and lowercase letters are different in Python. The letters displayed in the beehive diagram should all be uppercase, but the words in the English lexicon and the word list displayed on the screen are all lowercase. At some point, your code will have to apply the necessary case conversions.

Possible extensions

- *Generate the puzzle word.* In the Spelling Bee solver that you create for this assignment, the user is responsible for entering the seven-letter puzzle string. It would be fun to try and generate letter combinations that make a good puzzle. Puzzles must include at least one pangram but should probably not produce word lists that are too large. According to the website `https://nytbee.com`, the number of words in the *New York Times* puzzles has varied between 21 and 81, and the total number of points has ranged from 50 to 444. The *New York Times* also reduces the number of words by avoiding including the `S` character in the puzzles.
- *Implement the user interface.* The application described in this guide does not allow users to solve the puzzle on their own. On the *New York Times* site, you can type in letters on the keyboard and build up your own word list. The `pgl` library does allow you to listen for key events. If you call

  ```
  gw.add_event_listener("key", key_action)
  ```

  the graphics window will call `key_action` whenever a key is pressed, passing in a `KeyEvent` object for which the only useful method (analogous to the `get_x` and `get_y` for mouse events) is `get_key`, which returns the character that triggered the event. For standard characters, this will be a one-characters string like `"A"` or `"&"`. For special characters, `get_key` returns a symbolic representation of the key enclosed in angle brackets. For example, if you hit the RETURN key to indicate the end of a word entry, the `get_key` method gives back the string `"<RETURN>"`. You can play around with this feature to discover the symbolic names for other keys.
- *Implement the shuffle button.* The Spelling Bee implementation on the *New York Times* site includes a button that shuffles the letters in the outer hexagons. Doing so sometimes makes it easier to find the words.