0.1 PyLetterLizard: Python Letter Lizard Implementation

Python is a multipurpose, high-level programming language which is used in the software development world. Python is friendly and easy-to-learn [**?**], and supports object-oriented, structured, and functional programming (to a certain extent). It can be found in a variety of applications [**?**], such as Google, YouTube, applications by NASA, and the New York Stock Exchange. The friendly nature of Python makes it a very good tool to be used in an educational setting, and many find it useful as a first programming language to learn.

In the Python version of Letter Lizard (PyLetterLizard), we utilize three files, namely `letter_lizard.py`, `game.py`, and `config.py`. These files all work in tandem to allow PyLetterLizard to operate correctly. To run the application, one can type `python letter_lizard.py` at the command line. PyLetterLizard requires Python 2.7 to be installed, as well as the corresponding version of Pygame. Upon launching, the function `main()` of `letter_lizard.py` is called. Before this occurs, however, the files `config.py` and `game.py` are included. `Config.py` declares configuration values which affect the placement of various game objects, as well as declares a couple utility methods and objects. `Game.py` declares a class Game which stores the game state of a game in PyLetterLizard. We will discuss this module more in a later section.

When `main()` of `letter_lizard.py` executes, Pygame objects are instantiated, and some screen buttons are created. Then, the game enters an "infinite" loop, which does three things over and over: process user inputs/events; update game states; redraw the screen. We use the notion of a *game state*, which is a value which reflects several states that the game can be in. This allows us to know when to draw objects which represent normal gameplay, as opposed to drawing the splash screen, options screen, etc.

As mentioned previously, `game.py` contains the class Game, which stores the game state, and contains several methods which allow game state to be altered. The reason we decided to use a class to represent a game is because it allows for creation of new games quite easily, and separates the functionality of a Game into a discrete structure. This modularization greatly aids in our debugging process.

`Game.py` contains several methods, that `letter_lizard.py` utilizes to update and process game states. The method `process_letter` is called when a user types a letter, and we check to see if the letter exists in the puzzle word, and update the data structures correspondingly. Shuffle is called when a user hits the space bar, and this uses the random.shuffle function to randomly shuffle the puzzle word. This aids in the user's ability to find words in the puzzle. The method "draw" allows an instance of the class Game to draw itself onto the screen. The screen object is passed into the draw method as an argument. This methodology is good, since a game object can be in charge of drawing itself. This allows letter_lizard.py to not have to concern itself with how to draw a game...

it delegates this task to the game object.

0.2 Constructs of Python used

Listing:

```
def process_backspace(self):
    self.message = ""
    if (len(self.letters_guessed) >= 1):
        letter_to_delete =
            self.letters_guessed[len(self.letters_guessed) - 1]
        del self.letters_guessed[len(self.letters_guessed) - 1]
        self.puzzle_letters_displayed[self.puzzle_letters_displayed.index('')]
            = letter_to_delete
```

In the example above, we demonstrate several constructs of Python. We demonstrate the ability for a Python file to define a member function (method), where the function has "self" as its argument. We use the "del" command in Python to delete the last element of the array "letters_guessed". This syntax for deleting array elements is a bit different than other programming languages.

Bash

```
def __find_length_counts(self, words):
    word_lengths = [len(w) for w in words]
    return dict([(length, word_lengths.count(length)) for length
        in set(word_lengths)])
```

In the above example, we have a private method "__find_length_counts", which given an argument "words" (which is a list of words), it will return a mapping from the unique counts of letters for each word, to a count of how many words have that letter-count. For this method, we utilize some functional aspects of Python. We use list comprehensions to iterate over all the lengths of word_lengths. We transform word_lengths into a set, so that we only have the unique members. Then, we iterate over all the tuples in that set, and for each one, the tuple conists of a mapping from the length of the word to the count of how many words have that length. The dict() construct will transform this list of tuples into a dictionary. The above method is very useful for generating placeholders for the possible words that solve the puzzle.