Project 2: Autocorrected Typing Software

Download Project Materials

To get started, download project materials `project2.zip` from our QQ group if you don't have one. Below is a list of all the files you will see in the `project2.zip`. However, you only have to make changes to `cats/cats.py` in this project.

```
project2
  |-cats
  |  |-gui_files   # A directory of various things used by the web gui.
  |  |-data
  |  |  |-sample_paragraphs.txt    # A file containing text samples to be typed.
  |  |  |-common_words.txt         # A file containing common English words in
order of frequency.
  |  |  `-words.txt                # A file containing many more English words in
order of frequency.
  |  |-cats.py     # The typing test logic.
  |  |-gui.py      # A web server for the web-based graphical user interface
(GUI).
  |  |-ucb.py      # Utility functions for CS 61A projects.
  |  `-utils.py    # Utility functions for interacting with files and strings.
  `-project2.pdf   # Instructions for this project you must read
```

Introduction

> **Important submission notes**: This project has three phases. You have seven days for Phase 1 and seven more days for Phase 2 and 3. It doesn't mean that you are restricted to complete Phase 2 and 3 in just seven days. That is, you have 14 days in total for this project but the Phase 1 should be finished in the first 7 days. We recommend starting and finishing Phase 1 as soon as possible to give yourself adequate time to complete Phases 2 and 3, which can be more time consuming. Check the exact deadline on our OJ website.
>
> After completing any problems required in each phase, you need to submit your answer to `Contest 'cats: phase 1'` and `Contest 'cats: phase 2&3'` correspondingly on our OJ website to get your answer scored. We recommend that you submit **after you finish each problem** so that you can find bugs as soon as possible.

In this project, you will write a program that measures typing speed. Additionally, you will implement typing autocorrect, which is a feature that attempts to correct the spelling of a word after a user types it. This project is inspired by typeracer.

Suggestions

For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do **not** modify any other functions. Doing so may result in your code failing our framework. Also, please do **not** change any function signatures (names, argument order, or number of arguments).

Phase 1: Typing

> **Important submission notes**: After completing any problems required in this phase, you need to submit your answer to `Contest 'cats: phase 1'` on our OJ website to get your answer scored. Don't forget to check the deadline on the OJ website.

Problem 1 (100 pt)

Implement `choose`, which selects which paragraph the user will type. It takes a list of `paragraphs` (strings), a `select` function that returns `True` for paragraphs that can be selected, and a non-negative index `k`. The `choose` function return's the `k`th paragraph for which `select` returns `True`. If no such paragraph exists (because `k` is too large), then `choose` returns the empty string.

Problem 2 (200 pt)

Implement `about`, which takes a list of `topic` words. It returns a function that can be passed to `choose` as the `select` argument. The returned function takes a paragraph and returns a boolean indicating whether that paragraph contains any of the words in `topic`.

To make this comparison accurately, you will need to ignore case (that is, assume that uppercase and lowercase letters don't change what word it is) and punctuation.

Assume that all words in the `topic` list are already lowercased and do not contain punctuation.

> **Hint**: You may use the string utility functions in `utils.py`.

Problem 3 (100 pt)

Implement `accuracy`, which takes a `typed` paragraph and a `reference` paragraph. It returns the percentage of words in `typed` that exactly match the corresponding words in `reference`. Case and punctuation must match as well.

A *word* in this context is any sequence of characters separated from other words by whitespace, so treat "dog;" as all one word.

If a typed word has no corresponding word in the reference because `typed` is longer than `reference`, then the extra words in `typed` are all incorrect.

If `typed` is empty, then the accuracy is zero.

Problem 4 (100 pt)

Implement `wpm`, which computes the *words per minute*, a measure of typing speed, given a string `typed` and the amount of `elapsed` time in seconds. Despite its name, *words per minute* is not based on the number of words typed, but instead the number of characters, so that a typing test is not biased by the length of words. The formula for *words per minute* is the ratio of the number of characters (including spaces) typed divided by 5 (a typical word length) to the elapsed time in minutes.

For example, the string `"I am glad!"` contains three words and ten characters (not including the quotation marks). The words per minute calculation uses 2 as the number of words typed (because 10 / 5 = 2). If someone typed this string in 30 seconds (half a minute), their speed would be 4 words per minute.

Time to test your typing speed! You can use the command line to test your typing speed on paragraphs about a particular topic. For example, the command below will load paragraphs about cats or kittens. See the `run_typing_test` function for the implementation if you're curious (but it is defined for you).

```
python cats.py -t cats kittens
```

You can try out the web-based graphical user interface (GUI) using the following command.

```
python gui.py
```

Congratulations! You have finished Phase 1 of this project!

Phase 2: Autocorrect

> **Important submission notes**: After completing any problems required in the following two phases, you need to submit your answer to `Contest 'cats: phase 2&3'` on our OJ website to get your answer scored. Don't forget to check the deadline on the OJ website.

In the web-based GUI, there is an autocorrect button, but right now it doesn't do anything. Let's implement automatic correction of typos. Whenever the user presses the space bar, if the last word they typed doesn't match a word in the dictionary but is close to one, then that similar word will be substituted for what they typed.

Problem 5 (200 pt)

Implement `autocorrect`, which takes a `user_word`, a list of all `valid_words`, a `diff_function`, and a `limit`.

If the `user_word` is contained inside the `valid_words` list, `autocorrect` returns that word. Otherwise, `autocorrect` returns the word from `valid_words` that has the lowest difference from the provided `user_word` based on the `diff_function`. However, if the lowest difference between `user_word` and any of the `valid_words` is greater than `limit`, then `user_word` is returned instead.

A diff function takes in three arguments, which are the two strings to be compared (first the `user_word` and then a word from `valid_words`), as well as the `limit`. The output of the diff function, which is a non-negative number, represents the amount of difference between the two strings.

Assume that `user_word` and all elements of `valid_words` are lowercase and have no punctuation.

Important: if multiple strings have the same lowest difference according to the `diff_function`, `autocorrect` should return the string that appears first in `valid_words`.

> **Hint**: Try using `max` or `min` with the optional `key` argument.

Problem 6 (200 pt)

Implement `sphinx_swap`, which is a diff function that takes two strings. It returns the minimum number of characters that must be changed in the `start` word in order to transform it into the `goal` word. If the strings are not of equal length, the difference in lengths is added to the total.

Here are some examples:

```
>>> big_limit = 10
>>> sphinx_swap("nice", "rice", big_limit)    # Substitute: n -> r
1
>>> sphinx_swap("range", "rungs", big_limit)  # Substitute: a -> u, e -> s
2
>>> sphinx_swap("pill", "pillage", big_limit) # Don't substitute anything,
length difference of 3.
3
>>> sphinx_swap("roses", "arose", big_limit)  # Substitute: r -> a, o -> r, s ->
o, e -> s, s -> e
5
>>> sphinx_swap("rose", "hello", big_limit)   # Substitue: r->h, o->e, s->l, e-
>l, length difference of 1.
5
```

If the number of characters that must change is greater than `limit`, then `sphinx_swap` should return any number larger than `limit` and should minimize the amount of computation needed to do so.

These two calls to `sphinx_swap` should take about the same amount of time to evaluate:

```
>>> limit = 4
>>> sphinx_swap("roses", "arose", limit) > limit
True
>>> sphinx_swap("rosesabcdefghijklm", "arosenopqrstuvwxyz", limit) > limit
True
```

> **Important**: You may not use `while` or `for` statements in your implementation. Use recursion.

Try turning on autocorrect in the GUI. Does it help you type faster? Are the corrections accurate? You should notice that inserting a letter or leaving one out near the beginning of a word is not handled well by this diff function. Let's fix that!

Problem 7 (300 pt)

Implement `feline_fixes`, which is a diff function that returns the minimum number of edit operations needed to transform the `start` word into the `goal` word.

There are three kinds of edit operations:

1. Add a letter to `start`,
2. Remove a letter from `start`,
3. Substitute a letter in `start` for another.

Each edit operation contributes 1 to the difference between two words.

```
>>> big_limit = 10
>>> feline_fixes("cats", "scat", big_limit)       # cats -> scats -> scat
2
>>> feline_fixes("purng", "purring", big_limit)   # purng -> purrng -> purring
2
>>> feline_fixes("ckiteus", "kittens", big_limit) # ckiteus -> kiteus -> kitteus
-> kittens
3
```

We have provided a template of an implementation in `cats.py`. This is a recursive function with three recursive calls. One of these recursive calls will be similar to the recursive call in `sphinx_swap`.

You may modify the template however you want or delete it entirely.

If the number of edits required is greater than `limit`, then `feline_fixes` should return any number larger than `limit` and should minimize the amount of computation needed to do so.

These two calls to `feline_fixes` should take about the same amount of time to evaluate:

```
>>> limit = 2
>>> feline_fixes("ckiteus", "kittens", limit) > limit
True
>>> sphinx_swap("ckiteusabcdefghijklm", "kittensnopqrstuvwxyz", limit) > limit
True
```

Try typing again. Are the corrections more accurate?

```
python gui.py
```

Extensions: You may optionally design your own diff function called `final_diff`. Here are some ideas for making even more accurate corrections:

- Take into account which additions and deletions are more likely than others. For example, it's much more likely that you'll accidentally leave out a letter if it appears twice in a row.
- Treat two adjacent letters that have swapped positions as one change, not two.
- Try to incorporate common misspellings

Phase 3: Multiplayer

Typing is more fun with friends! You'll now implement multiplayer functionality, so that when you run `gui.py` on your computer, it connects to the course server at cats.cs61a.org and looks for someone else to race against.

To race against a friend, 5 different programs will be running:

- Your GUI, which is a program that handles all the text coloring and display in your web browser.
- Your `gui.py`, which is a web server that communicates with your GUI using the code you wrote in `cats.py`.
- Your opponent's `gui.py`.
- Your opponent's GUI.
- The CS 61A multiplayer server, which matches players together and passes messages around.

When you type, your GUI sends what you have typed to your `gui.py` server, which computes how much progress you have made and returns a progress update. It also sends a progress update to the multiplayer server, so that your opponent's GUI can display it.

Meanwhile, your GUI display is always trying to keep current by asking for progress updates from `gui.py`, which in turn requests that info from the multiplayer server.

Each player has an `id` number that is used by the server to track typing progress.

Problem 8 (200 pt)

Implement `report_progress`, which is called every time the user finishes typing a word. It takes a list of the words `typed`, a list of the words in the `prompt`, the user `id`, and a `send` function that is used to send a progress report to the multiplayer server. Note that there will never be more words in `typed` than in `prompt`.

Your progress is a ratio of the words in the `prompt` that you have typed correctly, up to the first incorrect word, divided by the number of `prompt` words. For example, this example has a progress of `0.25`:

```
report_progress(["Hello", "ths", "is"], ["Hello", "this", "is", "wrong"], ...)
```

Your `report_progress` function should return this number. Before that, it should send a message to the multiplayer server that is a two-element dictionary containing the keys `'id'` and `'progress'`. The `id` is passed into `report_progress` from the GUI. The progress is the fraction you compute. Call `send` on this dictionary to send it to the multiplayer server.

Problem 9 (100 pt)

Implement `time_per_word`, which takes in `times_per_player`, a list of lists for each player with timestamps indicating when each player finished typing each word. It also takes in a list `words`. It returns a `game` with the given information.

A `game` is a data abstraction that has a list of `words` and `times`. The `times` are stored as a list of lists of how long it took each player to type each word. `times[i][j]` indicates how long it took player `i` to type word `j`.

For example, if `times_per_player = [[1, 3, 5], [2, 5, 6]]`, the corresponding `time` attribute of the `game` would be `[[2, 2], [3, 1]]`. Timestamps are cumulative and always increasing, while the values in `time` are differences between consecutive timestamps.

Be sure to use the `game` constructor when returning a `game`, rather than assuming a particular data format.

Problem 10 (200 pt)

Implement `fastest_words`, which returns which words each player typed fastest. This function is called once both players have finished typing. It takes in a `game`.

The `game` argument is a `game` data abstraction, like the one returned in Problem 9. You can access words in the `game` with selectors `word_at`, which takes in a `game` and the `word_index` (an integer). You can access the time it took any player to type any word using `time`.

The `fastest_words` function returns a list of lists of words, one list for each player, and within each list the words they typed the fastest. In the case of a tie, consider the earliest player in the list (the smallest player index) to be the one who typed it the fastest.

Be sure to use the accessor functions for the `game` data abstraction, rather than assuming a particular data format.

Congratulations! Now you can play against other students in the course. Set `enable_multiplayer` to True near the bottom of cats.py and type swiftly!

```
python gui.py
```

Congratulations, you have reached the end of your second project! If you haven't already, relax and enjoy a few games of Cats with a friend.