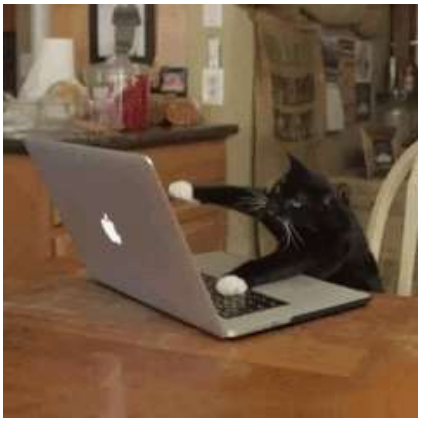
Project 2: CS 61A Autocorrected Typing Software

cats.zip (cats.zip)



Programmers dream of Abstraction, recursion, and Typing really fast.

Introduction

Important submission note: For full credit:

- Submit with Phases 1 and 2 complete by **Thursday, February 22**, worth 1 pt.
- Submit with all phases complete by **Tuesday, February 27**.

Try to attempt the problems in order, as some later problems will depend on earlier problems in their implementation and therefore also when running `ok` tests.

The entire project can be completed with a partner.

You can get 1 bonus point by submitting the entire project by **Monday, February 26**.

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 (https://play.typeracer.com/).

Final Product

Our staff solution to the project can be interacted with at cats.cs61a.org (https://cats.cs61a.org). If you'd like, feel free to try it out now. When you finish the project, you'll have implemented a significant part of this yourself!

Download Starter Files

You can download all of the project code as a zip archive (cats.zip). This project includes several files, but your changes will be made only to `cats.py`. Here are the files included in the archive:

- `cats.py`: The typing test logic.
- `utils.py`: Utility functions for interacting with files and strings.
- `ucb.py`: Utility functions for CS 61A projects.
- `data/sample_paragraphs.txt`: Text samples to be typed. These are scraped (https://github.com/kavigupta/wikivideos/blob/626de521e04ca643751ed85d549faca6ea528b1d/get_corpus.py) Wikipedia articles about various subjects.
- `data/common_words.txt`: Common English words in order of frequency (https://github.com/first20hours/google-10000-english/blob/master/google-10000-english-usa-no-swears.txt).
- `data/words.txt`: Many more English words in order of frequency (https://github.com/first20hours/google-10000-english/blob/master/google-10000-english-usa-no-swears.txt).
- `data/final_diff_words.txt`: Even more English words!
- `data/testcases.out`: Test cases for the optional Final Diff extension.
- `cats_gui.py`: A web server for the web-based graphical user interface (GUI).
- `gui_files`: A directory of files needed for the graphical user interface (GUI).
- `multiplayer`: A directory of files needed to support multiplayer mode.
- `favicons`: A directory of icons.
- `images`: A directory of images.
- `ok`, `proj02.ok`, `tests`: Testing files.
- `score.py`: Part of the optional Final Diff extension.

Logistics

The project is worth 20 points. 19 points are for correctness and 1 point is for submitting Phases 1 & 2 by the checkpoint date.

You will turn in the following files:

- `cats.py`

You do not need to modify or turn in any other files to complete the project. To submit the project, **submit the required files to the appropriate Gradescope assignment.**

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 or edit any files not listed above. Doing so may result in your code failing our autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an **autograder** called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to **log in with your Ok account using your web browser**. Please do so. Each time you run `ok`, it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations.

If you want to test your code interactively, you can run

```
python3 ok -q [question number] -i
```

with the appropriate question number (e.g. `01`) inserted. This will run the tests for that question until the first one you failed, then give you a chance to test the functions you wrote interactively.

You can also use the debugging print feature in OK by writing

```
print("DEBUG:", x)
```

which will produce an output in your terminal without causing OK tests to fail with extra output.

Getting Started Videos

> To see these videos, you should be logged into your berkeley.edu email.

Getting Started Videos

Phase 1: Typing

Problem 1 (1 pt)

Throughout the project, we will be making changes to functions in `cats.py`.

Implement `pick`. This function selects which paragraph the user will type. It takes three parameters:

- a list of paragraphs (strings) called `paragraphs`
- a `select` function, which returns `True` for paragraphs that can be selected
- a non-negative index `k`

The `pick` function returns the `k` th paragraph for which `select` returns `True`. If no such paragraph exists (because `k` is too large), then `pick` returns the empty string.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 01 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 01
```

Problem 2 (1 pt)

Implement `about`, which takes a list of `subject` words. It returns a function which takes a paragraph and returns a boolean indicating whether that paragraph contains any of the words in `subject`.

Once we've implemented `about`, we'll be able to pass the returned function to `pick` as the `select` argument, which will be useful as we continue to implement our typing test.

To be able 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 in the paragraph. Additionally, only check for exact matches of the words in subject in the paragraph, not substrings. For example, "dogs" is not a match for the word "dog".

> **Hint**: Use the `split`, `lower`, and `remove_punctuation` functions in `utils.py`.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 02 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 02
```

Problem 3 (2 pts)

Implement `accuracy`, which takes a `typed` paragraph and a `source` paragraph. It returns the percentage of words in `typed` that exactly match the corresponding words in `source`. Case and punctuation must match as well. "Corresponding" here means that two words must occur at the same indices in `typed` and `source`; the first words of both must match, the second words of both must match, and so on.

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

If `typed` is longer than `source`, then the extra words in `typed` that have no corresponding word in `source` are all incorrect.

If both `typed` and `source` are empty, then the accuracy is 100.0. If `typed` is empty but `source` is not empty, then the accuracy is zero. If `typed` is not empty but `source` is empty, then the accuracy is zero.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 03 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 03
```

Problem 4 (1 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 groups of 5 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 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.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 04 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 04
```

Time to test your typing speed! You can use the command line to test your typing speed on paragraphs about a particular subject. 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).

```
python3 cats.py -t cats kittens
```

You can try out the web-based graphical user interface (GUI) using the following command. (You may have to use `Ctrl+C` or `Cmd+C` on your terminal to quit the GUI after you close the tab in your browser).

```
python3 cats_gui.py
```

Phase 2: Autocorrect

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 (2 pts)

Implement `autocorrect`, which takes a `typed_word`, a `word_list`, a `diff_function`, and a `limit`. The goal of `autocorrect` is to return the word in `word_list` that is closest to the provided `typed_word`.

Specifically, `autocorrect` does the following:

- If the `typed_word` is contained inside the `word_list`, `autocorrect` returns that word.
- Otherwise, `autocorrect` returns the word from `word_list` that has the lowest difference from the provided `typed_word` based on the `diff_function`.
- However, if the lowest difference between `typed_word` and any of the words in `word_list` is greater than `limit`, then `typed_word` is returned instead. That is, `limit` puts a "limit" on how bad of a typo can be corrected.

Note: Assume that `typed_word` and all elements of `word_list` are lowercase and have no punctuation.

> **Important**: If multiple strings in `word_list` are tied for the lowest difference from `typed_word`, `autocorrect` should return the string that appears closest to the front of `word_list`.

A diff function takes in three arguments. The first is the `typed_word`, the second is the source word (in this case, a word from `word_list`), and the third argument is the `limit`. The output of the diff function, which is a number, represents the amount of difference between the two strings.

Here is an example of a diff function that computes the minimum of `1 + limit` and the difference in length between the two input strings:

```
>>> def length_diff(w1, w2, limit):
...     return min(limit + 1, abs(len(w2) - len(w1)))
>>> length_diff('mellow', 'cello', 10)
1
>>> length_diff('hippo', 'hippopotamus', 5)
6
```

> **Hint**: Try using `max` or `min` with the optional `key` argument (which takes in a one-argument function). For example, `max([-7, 2, -1], key = abs)` would return `-7` since `abs(-7)` is greater than `abs(2)` and `abs(-1)`.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 05 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 05
```

Problem 6 (3 pts)

Implement `feline_fixes`, which is a diff function that takes two strings. It returns the minimum number of characters that must be changed in the `typed` word in order to transform it into the `source` 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
>>> feline_fixes("nice", "rice", big_limit)    # Substitute: n -> r
1
>>> feline_fixes("range", "rungs", big_limit)  # Substitute: a -> u, e -> s
2
>>> feline_fixes("pill", "pillage", big_limit) # Don't substitute anything, length difference of 3.
3
>>> feline_fixes("goodbye", "good", big_limit) # Don't substitute anything, length difference of 3.
3
>>> feline_fixes("roses", "arose", big_limit)  # Substitute: r -> a, o -> r, s -> o, e -> s, s -> e
5
>>> feline_fixes("rose", "hello", big_limit)   # Substitute: r->h, o->e, s->l, e->l, length difference of
5
```

Important: You may not use `while`, `for`, or list comprehensions in your implementation. Use recursion.

If the number of characters that must change 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.

Why is there a limit? We know that `autocorrect` will reject any `source` word whose difference with the `typed` word is greater than `limit`. It doesn't matter if the difference is greater than `limit` by 1 or by 100; autocorrect will reject it just the same. Therefore, as soon as we know the difference will be above `limit`, it makes sense to try to minimize extra computation, even if the returned difference won't be exactly correct.

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

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

To ensure that you are correctly minimizing the amount of extra computation that is performed after the `limit` is reached, there is an autograder test that measures the performance of your solution based on the number of function calls that it makes. The test isn't perfect; using a helper function may cause this test to fail even if you are successfully avoiding extra computation.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 06 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 06
```

Try turning on autocorrect in the GUI. Does it help you type faster? Are the corrections accurate?

Problem 7 (3 pts)

Implement `minimum_mewtations`, which is a diff function that returns the minimum number of edit operations needed to transform the `typed` word into the `source` word.

There are three kinds of edit operations, with some examples:

1. Add a letter to `typed`.
    - Adding `"k"` to `"itten"` gives us `"kitten"`.
2. Remove a letter from `typed`.
    - Removing `"s"` from `"scat"` givs us `"cat"`.
3. Substitute a letter in `typed` for another.
    - Substituting `"z"` with `"j"` in `"zaguar"` gives us `"jaguar"`.

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

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

We have provided a template of an implementation in `cats.py`. You may modify the template however you want or delete it entirely.

> **Hint:** This is a recursive function with three recursive calls. One of these recursive calls will be similar to the recursive call in `feline_fixes`. Additionally, you will need more than one base case to solve this problem.

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

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

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

To ensure that you are correctly minimizing the amount of extra computation that is performed after the `limit` is reached, there is an autograder test that measures the performance of your solution based on the number of function calls that it makes.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 07 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 07
```

Try typing again. Are the corrections more accurate?

```
python3 cats_gui.py
```

Submit your Phase 1 and Phase 2 Checkpoint

Check to make sure that you completed all the problems in Phase 1 and Phase 2A:

```
python3 ok --score
```

Then, submit `cats.py` to the **Cats Checkpoint** assignment on **Gradescope** before the checkpoint deadline.

When you run `ok` commands, you'll still see that some tests are locked because you haven't completed the whole project yet. You'll get full credit for the checkpoint if you complete all the problems up to this point.

(Optional) Extension: Final Diff (0 pts)

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.

You can also set the limit you'd like your diff function to use by changing the value of the variable `FINAL_DIFF_LIMIT` in `cats.py`.

You can check your `final_diff`'s success rate by running:

```
python3 score.py
```

If you don't know where to start, try copy-pasting your code for `feline_fixes` and `minimum_mewtations` into `final_diff` and scoring them. Looking at the typos they accidentally fixed might give you some ideas!

Phase 3: Multiplayer

Typing is more fun with friends! You'll now implement multiplayer functionality, so that when you run `cats_gui.py` on your computer, it connects to the course server at cats.cs61a.org (https://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 `cats_gui.py`, which is a web server that communicates with your GUI using the code you wrote in `cats.py`.
- Your opponent's `cats_gui.py`.
- Your opponent's GUI.
- The CS 61A multiplayer server, which matches players together and passes messages around.

When you type, your GUI uploads what you have typed to your `cats_gui.py` server, which computes how much progress you have made and returns a progress update. It also uploads 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 `cats_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 (2 pts)

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 `source`, the user's `user_id`, and a `upload` function that is used to upload a progress report to the multiplayer server. There will never be more words in `typed` than in `source`.

Your progress is a ratio of the words in the `source` that you have typed correctly, up to the first incorrect word, divided by the number of `source` 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 do two things: upload a message to the multiplayer server and return the progress of the player with `user_id`.

You can upload a message to the multiplayer server by calling the `upload` function on a two-element dictionary containing the keys `'id'` and `'progress'`. You should then return the player's progress, which is the ratio of words you computed.

> **Hint:** See the dictionary below for an example of a potential input into the `upload` function. This dictionary represents a player with `user_id` 1 and `progress` 0.6.
>
> `{'id': 1, 'progress': 0.6}`

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 08 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 08
```

Problem 9 (2 pts)

Implement `time_per_word`, which takes in a list `words` and `timestamps_per_player`, a list of lists for each player with timestamps indicating when each player finished typing every individual word in `words`. It returns a `match` with the given information.

A `match` is a *data abstraction* that represents a typing "match" between multiple players. Speficially, each `match` stores the instance variables `words` and `times`.

- The `times` are stored as a list of lists of how long it took each player to type every word in `words`.
- Specifically, `times[i][j]` indicates how long it took player `i` to type `words[j]`.

For example, say `words = ['Hello', 'world']` and `times = [[5, 1], [4, 2]]`, then `[5, 1]` corresponds to the list of times for player 0, and `[4, 2]` corresponds to the list of times for player 1. Thus, player 0 took `5` units of time to write the word `'Hello'`.

> **Important**: Be sure to use the `match` constructor when returning a `match`. The tests will check that you are using the `match` data abstraction rather than assuming a particular data format.
>
> For more information, you can read the definitions for the `match` constructor below or in `cats.py`. However, as with any data abstraction, we are only concerned with what our functions do rather than their specific implementations!

Match Data Abstraction

Timestamps are cumulative and always increasing, while the values in `times` are **differences between consecutive timestamps for each player**.

Here's an example: If `timestamps_per_player = [[1, 3, 5], [2, 5, 6]]`, the corresponding `times` attribute of the `match` would be `[[2, 2], [3, 1]]`. This is because the differences in timestamps are `(3-1)`, `(5-3)` for the first player and `(5-2)`, `(6-5)` for the second player. The first value of each list within `timestamps_per_player` represents the initial starting time for each player.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 09 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 09
```

Problem 10 (2 pts)

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

Specifically, 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 (against all the other players). 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.

For example consider the following match with the words `'Just'`, `'have'`, and `'fun'`. Player 0 typed `'fun'` the fastest (3 seconds), Player 1 typed `'Just'` the fastest (4 seconds), and they tied on the word `'have'` (both took 1 second) so we consider to Player 0 to be the fastest, because they are the earliest player in the list.

```
>>> player_0 = [5, 1, 3]
>>> player_1 = [4, 1, 6]
>>> fastest_words(match(['Just', 'have', 'fun'], [player_0, player_1]))
[['have', 'fun'], ['Just']]
```

The `match` argument is a `match` data abstraction, just like the one we returned in Problem 9.

- You can access words in a `match` with the selector `get_word`, which takes in a `match` and the `word_index` (an integer).
- In addition, you can access the time it took a player to type the word at a particular index using the `time` function, which takes takes an integer `player_num`, in addition to `match` and `word_index`.
- With these two functions and a `match`, we can easily access the time it took any player to type any word!

```
>>> player_0 = [5, 1, 3]
>>> player_1 = [4, 1, 6]
>>> ex_match = match(['Just', 'have', 'fun'], [player_0, player_1])
>>> get_word(ex_match, 2)
'fun'
>>> time(ex_match, 0, 2)
3
```

Important: Be sure to use the `match` selectors when using a `match`. The tests will check that you are using the `match` data abstraction rather than assuming a particular data format.

Make sure your implementation does not mutate the given player input lists. For the example above, calling `fastest_words` on `[player_0, player_1]` should **not** mutate `player_0` or `player_1`.

Before writing any code, unlock the tests to verify your understanding of the question:

```
python3 ok -q 10 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 10
```

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!

```
python3 cats_gui.py
```

Project Submission

Run `ok` on all problems to make sure all tests are unlocked and pass:

```
python3 ok
```

You can also check your score on each part of the project:

```
python3 ok --score
```

Once you are satisfied, submit this assignment by uploading `cats.py` to the **Cats** assignment on **Gradescope.** For a refresher on how to do this, refer to Lab 00 (/~cs61a/sp24/lab/lab00/#task-c-submitting-the-assignment).

You can add a partner to your Gradescope submission by clicking on **+ Add Group Member** under your name on the right hand side of your submission. Only one partner needs to submit to Gradescope.