Autocomplete

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

The following file contains code and other resources as a starting point for this lab: `autocomplete.zip`

Note that passing all of the tests on the server will require that your code runs reasonably efficiently.

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- answering the questions on this page (1 point)
- passing the style check (1 point)
- passing the tests in `test.py` (3 points)

> **Reminder: Academic Integrity**
>
> Please also review the academic-integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

2) Introduction

Type "aren't you" into a search engine and you'll get a handful of search suggestions, ranging from "aren't

you clever?" to "aren't you a little short for a stormtrooper?". If you've ever done a web search, you've probably seen an autocompletion — a handy list of words that pops up under your search, guessing at what you were about to type.

Search engines aren't the only place you'll find this mechanism. Just to name a few: cell phones use autocomplete/autocorrect to predict/correct words that are entered for, for example, text messages; and some IDEs use autocomplete to make the process of coding more efficient by offering suggestions for completing long function or variable names.

In this lab, we are going to implement our own version of an autocomplete/autocorrect engine using a tree structure called a *prefix tree*, as described in this document, and then we'll use that structure to perform some text-processing tasks.

> **Note**
>
> At several points throughout this lab, you will be asked to write some test cases of your own. We strongly encourage you to try writing those tests *before* implementing the associated behavior (by thinking through what the output should be on some small examples), so that you can use them to help with testing and debugging when you are ready to implement the associated functionality.
>
> Of course, if you have trouble thinking about how to structure your test cases, what test cases might be useful, etc., please don't hesitate to ask for help!

2.1) Prefix Trees

A prefix tree is a type of tree that stores an associative array (a mapping from keys to values). The prefix tree stores keys organized by their prefixes (their first characters), with longer prefixes given by successive levels of the prefix tree. Each node optionally contains a value to be associated with that node's prefix. Prefix trees are often used in text-processing applications, and so we'll limit our attention to prefix trees whose keys are given by strings.

As an example, consider a prefix tree constructed as follows:

```
t = PrefixTree()
t['bat'] = 7
t['bar'] = 3
t['bark'] = ':)'
```

This prefix tree would look like the following (Fig. 1).

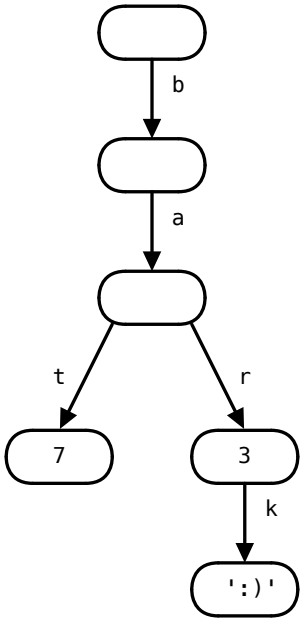

Fig. 1

Two important things to notice here:

1. There is no single object that represents the whole structure of this prefix tree. Rather, each node in the diagram above is represented by a single `PrefixTree` instance.

2. The keys associated with the nodes are not actually stored in the nodes themselves. Rather, they are associated with the edges connecting the nodes.

We'll start by implementing a class called `PrefixTree` to represent prefix trees in Python. This class will include facilities for adding, deleting, modifying, and iterating over key/value pairs. For example, consider the following example:

```
>>> t = PrefixTree()
>>> t['bat'] = True
>>> t['bar'] = True
>>> t['bark'] = True
>>>
>>> t['bat']
True
>>> t['something']
Traceback (most recent call last):
    ...
KeyError
>>>
>>> t['bark'] = 20
>>> t['bark']
20
>>>
>>> for i in t:
```

```
        print(i)


('bat', True)
('bar', True)
('bark', 20)
>>>
>>> del t['bar']
>>>
>>> for i in t:
        print(i)


('bat', True)
('bark', 20)
```

Note that, in terms of interface and functionality, the `PrefixTree` class will have a lot in common with a Python dictionary. However, the representation we're using "under the hood" has some nice features that make it particularly well-suited for tasks that use prefix-based lookups (such as autocompletion).

3) `PrefixTree` Class and Basic Methods

In `lab.py`, you are responsible for implementing the `PrefixTree` class, which should support the following methods.

__init__(*self*)

The `__init__` method takes no arguments. It should set up exactly two instance variables:

- `value`, the value associated with the sequence ending at this node. Initial value is `None` (we will assume that a value of `None` means that a given key has no value associated with it, not that the value `None` is associated with it).
- `children`, a dictionary mapping a single-element sequence (in our case, a length-1 string) to another node, i.e., the next level of the prefix-tree hierarchy (prefix trees are a recursive data structure). Initial value is an empty dictionary.

How many `PrefixTree` instances comprise the example structure in the drawing above (*Figure 1*)?

The top node in *Figure 1* is represented by a `PrefixTree` instance; let's call it `t`. In that case, `t.children` represents a dictionary. What are the keys in this dictionary? Enter a list or tuple containing all of the keys:

What is the type of the values in the `t.children` dictionary?

- `str`
- `list`
- `tuple`
- `dict`
- `PrefixTree`
- something else

`__setitem__`(*self, key, value*)

Add the given `key` to the prefix tree, associating it with the given `value`. For the prefix-tree node that marks the end of the key, set that node's `value` attribute to the given `value` argument. This method doesn't return a value. This is a special method name used by Python to implement subscript assignment. For example, `x[k] = v` is translated by Python into `x.__setitem__(k, v)`. If the given `key` is not a string, a `TypeError` exception should be raised (see Python documentation here).

Examples (using the structure from the picture above (*Figure 1*)):

- `t = PrefixTree()` would create the root node of the example above.
- `t['bat'] = 7` adds three nodes (representing the `'b'`, `'ba'`, and `'bat'` prefixes) and associates the value `7` with the node corresponding to `'bat'`.
- `t['bark'] = ':)'` adds two new nodes for prefixes `'bar'` and `'bark'` shown on the bottom right of the prefix tree, setting the value of the last node to `':)'`.
- `t['bar'] = 3` doesn't add any nodes and only sets the value of the first node added above when inserting "bark" to `3`.
- `t[1] = True` raises a `TypeError` (because the given key is not of the appropriate type) and does not make any changes to the prefix tree.

Continuing the example from *Figure 1* above, how many new `PrefixTree` instances will be created if we execute `t['bank'] = 4`?

If we then run `t['ban'] = 7`, how many new `PrefixTree` instances will be created?

`__getitem__`(*self, key*)

Return the value associated with the given key. This is a special method name used by Python to implement subscripting (indexing). For example, `x[k]` is translated by Python into `x.__getitem__(k)`. Your code should find the corresponding node for the given `key` and return the associated value. You should raise a `KeyError` if the key cannot be found in the prefix tree. If the given key is not a string, raise a `TypeError` instead.

Examples (using the first example from *Figure 1* above):

- `t['bark']` should return `':)'`.
- `t['apple']` should raise a `KeyError` since the given key does not exist in the prefix tree.
- `t['ba']` should also raise a `KeyError` since, even though the key `'ba'` is represented by a node in the prefix tree, it has no value associated with it.
- `t[1]` should raise a `TypeError` since `1` is not a string.

Check Yourself:

Add some test cases (either as doctests or in `test.py`) for `__getitem__` to test that it is working as expected. What cases are important to test?

`__delitem__`(*self, key*)

Disassociate the given key from its value. This is a special method name used by Python to implement index deletion. For example, `del x[k]` is translated by Python into `x.__delitem__(k)`. Trying to delete a key that has no associated value should raise a `KeyError`, and trying to delete a key that isn't a string should raise a `TypeError`.

Examples (using the example from *Figure 1* above):

- `del t["bar"]` should disassociate `"bar"` from its value, so that any subsequent lookup of `t["bar"]` produces a `KeyError`.
- `del t["foo"]` should raise a `KeyError` since `"foo"` does not exist as a key in the example prefix tree.

For the purposes of this lab, you only need to do the bare minimum so that the key is no longer associated with a value (don't worry about extra memory usage after deleting the key). As an interesting optional feature (and a nice improvement to the structure), you could also try removing all unnecessary nodes from the prefix-tree structure.

Check Yourself:

Add some test cases for `__delitem__` to make sure it is working as expected. What cases are important to test?

__contains__(*self, key*)

Return `True` if `key` occurs and has a value other than `None` in the prefix tree. Raise a `TypeError` if the given key is not a string. `__contains__` is the special method name used by Python to implement the `in` operator. For example, `k in x` is translated by Python into `x.__contains__(k)`.

Hint: At first glance, the code for this method might look very similar to some of the other methods above. Make good use of helper functions to avoid repetitious code!

What should be the result of evaluating `'bar' in t`, using the example from *Figure 1* above?

What should be the result of evaluating `'barking' in t`, using the example from *Figure 1* above?

What should be the result of evaluating `'ba' in t`, using the example from *Figure 1* above?

Check Yourself:

Add some tests for `__contains__` to make sure it is working as expected. What cases are important to test?

__iter__(*self*)

`__iter__` should be a generator that yields a `(key, value)` tuple for each key stored in the prefix tree. The pairs can be produced in any order. `__iter__` is the special method name used by Python when it needs to iterate over a data object, i.e., the method invoked by the `iter()` built-in function. For example, the following Python code will print all the keys in a prefix tree:

```
for key, val in t:
```

```
    print(key)
```

You should do this *without* first making a list or other structure that contains the items in the prefix tree.

Hint: You'll want to write a recursive generator function that uses `yield` (and maybe `yield from`) to produce the required sequence of values one at a time. See the Python documentation for generators and/or yield from.

Examples (using *Figure 1* from above):

- The example above prints `bat`, `bar`, and `bark` on three separate lines.
- `list(t)` returns `[('bat', 7), ('bar', 3), ('bark', ':)')]` (but not necessarily in that order). Note that the `list` function has an internal `for` loop that uses `iter(t)` to iterate over each element of the sequence `t`.

4) Autocomplete

Now that we've implemented the skeleton of the prefix-tree structure itself, let's implement our autocomplete engine! To this end, we'll need to build up a particular kind of prefix tree based on a piece of text: a *frequency* prefix tree, whose keys are words and whose values are the numbers of times that those words occurs in the given text.

We'll implement this as a function `word_frequencies`, described below:

word_frequencies(*text*)

`text` is a string containing a body of text. Return a `PrefixTree` instance mapping *words* in the text to the frequency with which they occur in the given piece of text.

Note that we have provided a method called `tokenize_sentences` which will try to intelligently split a piece of text into individual sentences. **You should use this function rather than implementing your own.** The function takes in a single string and returns a list of sentence strings. Each sentence string has had its punctuation removed, leaving only the words. Words within the sentence strings are sequences of characters separated by spaces.

Check Yourself:

Add some tests for `word_frequencies`, including at least one example of your own devising.

Check Yourself:

If you work just by using `__getitem__` and `__setitem__` as described above, your code will likely run somewhat slowly, since incrementing the number associated with a word involves first traversing part of the prefix tree to find the associated count and then traversing it again to find the node a second time to increment its value. Can you think of a way to improve the efficiency here (perhaps by adding additional features to the prefix-tree representation)?

As a running example, we'll use the following structure (Fig. 2), which could have been created by calling `word_frequencies("bat bat bark bar")`.

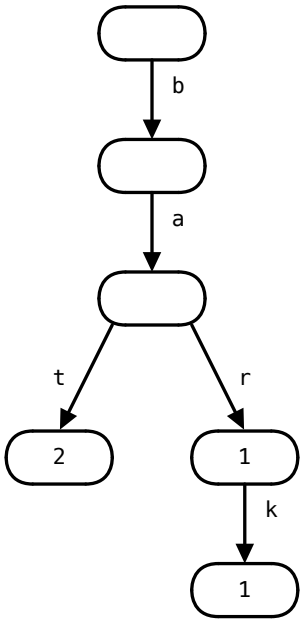

Fig. 2

Once we have this representation, we are ready to go ahead and implement autocompletion! We'll implement autocompletion as a function described below:

autocomplete(*tree, prefix, max_count*=None)

> `tree` is an instance of `PrefixTree`, `prefix` is a string, `max_count` is an integer or `None`. Return a list of the `max_count` most-frequently-occurring keys that start with `prefix`. In the case of a tie, you may output any of the most-frequently-occurring keys. If there are fewer than `max_count` valid keys available starting with `prefix`, return only as many as there are. The returned list may be in any order. If `max_count` is not specified, your list should contain *all* keys that start with `prefix`.
>
> Return `[]` if `prefix` is not in the prefix tree. Raise a `TypeError` if the given prefix is not a string.

If `t` refers to the example structure in *Figure 2*, what should be the result of evaluating the following Python expression?

```
autocomplete(t, "ba", 1)
```

With that same example, there are multiple possible correct ways that the following expression could be evaluated:

```
autocomplete(t, "ba", 2)
```

Enter any one such valid result in the box below:

Using the same example, what should be the result of evaluating the following?

```
autocomplete(t, "be", 2)
```

Importantly, the structure of the prefix tree allows us to implement autocomplete efficiently, **without needing to consider all of the values in the entire prefix tree**.

Check Yourself:

Write a few small tests of your own to test this behavior. You may include the above examples as test cases, but you should also include at least 1 nontrivial test case of your own devising.

5) Autocorrect

You may have noticed that for some words, our autocomplete implementation generates very few or no suggestions. In cases such as these, we may want to guess that the user mistyped something in the original word. We ask you to implement a more sophisticated tool: autocorrect.

autocorrect(*tree, prefix, max_count*=None)

`tree` is an instance of `PrefixTree`, `prefix` is a string, `max_count` is an integer or `None`. It returns a list of up to `max_count` words. `autocorrect` should invoke `autocomplete` but, if fewer than `max_count` completions are made, suggest additional words by applying one **valid edit** to the prefix.

An **edit** for a word can be any one of the following:

- A single-character insertion (add any one character in the range "a" to "z" at any place in the word)
- A single-character deletion (remove any one character from the word)
- A single-character replacement (replace any one character in the word with a character in the range "a" to "z")
- A two-character transpose (switch the positions of any two adjacent characters in the word)

A **valid edit** is an edit that **results in a word in the prefix tree without considering any suffix characters**. In other words, we don't try to autocomplete valid edits, we just check whether the edit exists in the prefix tree.

For example, editing `"te"` to `"the"` is valid, but editing `"te"` to `"tze"` is not, as "tze" isn't a word. Likewise, editing `"phe"` to `"the"` is valid, but `"phe"` to `"pho"` is not because "pho" is not a word in the corpus, although many words beginning with "pho" are.

In summary, given a prefix that produces C completions, where $C <$ `max_count`, generate up to `max_count` $-C$ additional words by considering all valid single edits of that prefix (i.e., corpus words that can be generated by 1 edit of the original prefix) and selecting the most-frequently-occurring edited words. Return a list of suggestions produced by including **all** C of the completions and up to `max_count` $-C$ of the most-frequently-occuring valid edits of the prefix; the list may be in any order. **Be careful not to repeat suggested words!**

If `max_count` is `None` (or is unspecified), `autocorrect` should return all autocompletions as well as all valid edits.

Example (using the example from *Figure 1* above):

- `autocorrect(t, "bar", 3)` returns `['bar', 'bark', 'bat']` since "bar" and "bark" are found by autocomplete, and "bat" is a valid edit involving a single-character replacement, i.e., "t" is replacing the "r" in "bar".

Check Yourself:

Write a few small tests of your own to test this behavior. You may include the above examples as test cases, but you should also include at least 1 nontrivial test case of your own devising.

6) Filtering Words

It's sometimes useful to select only the words from a corpus that match a pattern. That's the purpose of the `word_filter` method.

word_filter(*tree, pattern*)

> `tree` is an instance of `PrefixTree`, and `pattern` is a string. Return a list of `(word, freq)` tuples that represent all the unique words in the `tree` whose characters match those of `pattern`. The list can be in any order.
>
> The characters in `pattern` are interpreted as follows:
>
> - `'?'` matches **exactly one** of the next unmatched characters in word no matter what it is. There must be a next unmatched character for `'?'` to match.
> - `'*'` matches a sequence of **zero or more** of the next unmatched characters in word.
> - Otherwise the character in the `pattern` must exactly match the next unmatched character in the word.
>
> Note that the characters replaced by `*` and `?` can be arbitrary characters, not just letters.
>
> Pattern examples:
>
> - `"year"` would only match with the word "year".
> - `"year?"` would match words such as "years" and "yearn" (but not longer words).
> - `"???"` would match all 3-letter words.
> - `"?ing"` matches all 4-letter words ending in "ing."
> - `"*ing"` matches all words ending in "ing."
> - `"?*ing"` matches all words with 4 or more letters that end in "ing."
> - `"*a*t"` matches any word that contains an "a" and ends in "t." This would include words like "at", "art", "saint", and "what."
> - `"year*"` would match words such as "year," "years," and "yearn," among others (as well as longer words like "yearning").
>
> Filter examples (using the example from Figure 2 above):
>
> - `word_filter(t, "bat")` returns `[('bat', 2)]`.
> - `word_filter(t, "ba?")` returns `[('bat', 2), ('bar', 1)]`, i.e., listing all the 3-letter words in the prefix tree that begin with "ba".
> - `word_filter(t, "*")` returns `[('bat', 2), ('bar', 1), ('bark', 1)]`, i.e., listing all the words in the prefix tree.
> - `word_filter(t, "*k")` returns `[('bark', 1)]`, i.e., listing all the words in the prefix tree that end with the letter k.

If `t` refers to the example structure in *Figure 2*, what should be the result of evaluating the following Python expression?

```
word_filter(t, "ba")
```

If `t` refers to the example structure in *Figure 2*, what should be the result of evaluating the following Python expression?

```
word_filter(t, "???")
```

If `t` refers to the example structure in *Figure 2*, what should be the result of evaluating the following Python expression?

```
word_filter(t, "*r*")
```

One way to break this problem down is to start by handling any `pattern` that does not include `'?'` or `'*'`. After you have implemented this behavior correctly, the `test_filter_word` test case should pass.

Next, add code that can handle the `'?'` character in the `pattern` as well. After this is working, `test_filter_question` should pass.

Finally, after correctly adding the `'*'` behavior, all the remaining `test_filter` test cases should pass!

Check Yourself:

Take a moment to plan out your approach before you start writing code. How can you efficiently make use of the structure of the given `tree` to find the words that match the given `pattern`? If you get stuck, there is an additional hint below!

Show/Hide

Hint: this operation can be implemented as a recursive search function that attempts to match the given `pattern` to the words in the given `tree` one character at a time. **You should not use a "brute-force" method that involves generating all possible words that could result from the given `pattern` and/or looping over all words in the `tree`.**

Note

You cannot use any of the built-in Python pattern-matching functions, e.g., functions from the `re` module — you are expected to write your own pattern-matching code. Copying or referencing code from StackOverflow or other sources is also not appropriate.

7) Testing your lab

As in the previous labs, we provide you with a `test.py` script to help you verify the correctness of your code. In addition to the test cases for this week's lab, we'll have you test out your code by running it on several examples of real public-domain books (courtesy of Project Gutenberg).

Download the following text files (each of which contains a whole book) to the same directory as your `lab.py` (by right-clicking the link and then clicking "Save As" or similar):

- *Pride and Prejudice* by Jane Austen
- *Alice's Adventures in Wonderland* by Lewis Carroll
- *Dracula* by Bram Stoker
- *A Tale of Two Cities* by Charles Dickens
- *Metamorphosis* by Franz Kafka

You can load the text of any of these files using something like the following code:

```
with open("filename.txt", encoding="utf-8") as f:
    text = f.read()
```

After running this code, the variable `text` will be bound to a string containing the text contained in the `filename.txt` file.

We'll read the contents of these files into Python, use our `word_frequencies` function to create the relevant prefix tree structures, and use our autocompletion/autocorrection based on those corpora. Use these tools to answer the following questions.

In *Metamorphosis*, what are the six most common words starting with `gre`? Enter your answer as a Python list of strings:

In *Metamorphosis*, what are all of the words matching the pattern `c*h`, along with their counts? Enter your answer as a Python list of tuples, of the same form as your output from `word_filter`:

In *A Tale of Two Cities*, what are all of the words matching the pattern `r?c*t`, along with their counts? Enter your answer as a Python list of tuples, of the same form as your output from `word_filter`:

What are the top 12 autocorrections for `'hear'` in *Alice in Wonderland*? Enter your answer as a Python list of strings:

What are *all* autocorrections for `'hear'` in *Pride and Prejudice*? Enter your answer as a Python list of strings:

How many distinct words are in *Dracula*?

How many total words are in *Dracula*?

8) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a autocomplete /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

> When this page was loaded, you had not yet made any submissions.
> If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.