Anki Autolookup

This is a final project from a programming course. It may not be maintained, but feel free to contact us if you want to contribute or maintain this project.

Table of Contents

Installation

This program is only tested on Ubuntu 19.04 and Ubuntu 19.10. To use it you need to clone the project first:

```
git clone https://github.com/fhcwcsy/anki-autolookup.git
```

You will also need an additional module **Tesseract**, please follow the installation there. For Ubuntu users, all you need to do is:

```
sudo apt install tesseract-ocr
sudo apt install libtesseract-dev
```

Then:

```
pip install pytesseract
```

You will also need two Anki plugins: `Allows empty first field (during adding and import)` and `AnkiConnect`. Add them by clicking `Tools - Add-ons` in Anki, then use `46741504` and `2055492159` to add the plugins.

Usage

Use:

```
./start.sh
```

to start the program. Anki must be running in the background so the API can work. Select the deck you want to add cards to, then choose a method to add words. Currently, three features are available:

- Word lookup: Simply type the words you want to lookup. The program will record the word whenever you press `<Return>`. The words will be shown in the wordlist on the right. You can uncheck the checkbuttons beside a word you don't want to add to your deck. After the lookup finishes, (the queue is empty), each word (if it is checked and **the lookup succeed**), the program will generate a card to the deck.
- Article lookup: Paste an article to the textbox, then click `Lookup!`. The program will find the difficult words in the article and list them in the wordlist. The wordlist works the same way as the previous feature. For the definition of *difficult words*, see the explanation of `article_lookup.py`.
- Image lookup: Choose a image (**must not be tilted or twisted or it will not be accurate.** Some examples are provided in `./main/imgexample/`.) Click on the words you want to lookup. The wordlist works the same way as the first feature.

Note that when you close the window of a feature, the main menu will show up **after the queue is cleared.**

Demonstration

Youtube video

File Description

```
.
+-- README.md: This documentation.
+-- start.sh: bash script to launch the program.
+-- main: Main scripts of the program.
|   +-- add_card.py: Functions to generate new cards and add to Anki.
|   +-- article_lookup.py: Lookup difficult words in an article.
|   +-- crawler.py: A cralwer to lookup words in Cambridge online dictionary.
|   +-- dic.xlsx: A spreadsheet with data about word usage in a variety of
sources.
|   +-- imgrecog.py: The script for word lookup from images.
|   +-- main.py: Main menu GUI.
|   +-- wordlist_cls.py: The script defining the wordlist window.
|   +-- word_lookup.py: GUI for adding inserted words to Anki.
+-- presentation: Slides used (Tex and pdf files) for presentation in class.
|   +-- proposal.*: Slides for the proposal.
+-- trash: scripts that are no longer in use.
```

Below we lists detailed descriptions for each script. **All of these are copied from the docstrings. You can simply skip these and read the codes if you want to understand the program. They are only listed because we are asked to do so.**

`add_card.py`

This file defines a Request class and three functions (`add_note`, `create_model`, `new_deck_name`), to connect to anki API and convert the information obtained by `crawler.py` to cards in anki.

The required modules are:

- json
- urllib.request
- collections (namedtuple)

Below we list the classes and functions in this file.

Request

A class to connect with anki API. This class is mainly copied from the anki-connect website.

Attributes:

- `_action`: A string, the action users want to do with anki API.
- `_request`: A dictionary, the request to anki API.
- `_response`: A dictionary, the response returned by anki API, associated with the request. Contains "result" and "error".
- `_result`: The result part of `_response`.

Class Methods

- `__init__(self)`

Constructor of the class.

```
Args:
    action: The action users want to do with anki API.
    params: The other necessary information associated with the action.

Returns:
    None

Raises:
    None
```

- `_invoke(self)`

This method connects with anki API and check if there is any mistake. If not, put the response in the attribute `_response`.

```
Args:
    None

Returns:
    None

Raises:
    Raise Exception ("response has an unexpected number of fields") if
        the anki-connect system returned an invalid response.
    Raise Exception ("response is missing required error field") if the
        anki-connect system didn't returned error field.
    Raise Exception ("response is missing required result field") if the
        anki-connect system didn't returned result field.
    Rasie Exception (self._response) if there exists errors in the
```

```
        returned response.
```

add_note(wordinfo)

This function calls `Request`, using `addnote` as action, `deckName` as deckname, `my_model` as modelname, `fields` as field. `deckname` is indicated by users. `my_model` is created by the function `create_model`. `fields` is made by the information got by `crawler.py`. This function rearrange the information got by `crawler.py` to fit the model, then create a new card in anki.

```
Args:
    wordinfo: A list of Entry objects obtained by the crawler. The format is
    demonstrated in `crawler.py`.

Returns:
    None

Raises:
    None
```

`create_model()`

This function creates a card model using html. Using class `Request` and action `modelName` to check if `my_model` is one of the user's deck name. If not, create one using class `Request` and action `createModel`.

```
Args:
    None

Returns:
    None

Raises:
    None
```

`_new_deck_name`

Update `deckName`. If `name` is an empty string as default, return None.

```
Args:
    name: A list, the deck names in the user's anki.

Returns:
    If name is '' (default), return None.
    If name is a list, return True.

Raises:
    None
```

`article_lookup.py`

This program will create a lookup window where the user can enter/paste an article and choose the difficulty of words he wants to be looked up. The program will find the words in the article that match the requirements and list in the wordlist window on his right. Then, user can choose the words he want to add to anki.

The *difficulty* of a word is defined as follows: We use a wordlist from NGSL, which contains the words used in fictions, journals, TV subtitles, etc., and their times of being used. Hence, the difficulties of a word is defined by the frequency of being used. If a word is more oftenly used, it is considered to be less difficult, and vice versa. If a word is not in the list, it is considered to be too difficult or rarely used for a foreign English learner to learn. The frequency used in this program is normalized by dividing the actual count of usage by the number of times used of the most frequently used word (so the frequency is between 0 and 1). This program reads the text paste in the textbox, determine the difficulty of each word by looking up in the local wordlist, then lookup each word in the Cambridge dictionary.

The required modules are:

- `tkinter`
- `re` (regular expression)
- `openpyxl`

Below we list all the classes in this file.

ArticleRecognitionWindow

This is the main window of the feature.

Attributes

- `difficulty`: The (normalized) maximum difficulty of the word to be looked up. The lower this value is, the less word (keeping only the most difficult ones) will be looked up.
- `_inputWindow`: A `tk.Toplevel()` object. The window on the left that allows the user to paste an article to be looked up.
- `_wordwindow`: A `tk.Toplevel()` object. The window on the right that allows the user to select the words they want to add to Anki.
- `_lookupButton`: A `tk.Button` object. The button that will analyze the article when pressed.
- `_inputBox`: A `tk.Text` object. The textbox that allows the user to paste text.
- `_wordlistFrame`: A `WordlistWindow` object. Shows the list of words found in the article with checkboxes.

Class Methods

- `__init__()`

Create two windows.

One is `_inputWindow` where the users enter the article and determine the difficulty of words they want to be looked up.
The other is `_wordwindow` where we show the the difficult words we find in the article. Users can select which words they want to make vocabulary cards here.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `lookup()`

This function will first get the difficulty users set. Use it to find the difficult words in the article, and add it to the `_wordwindow`.

```
Args:
    None

Return:
    None

Raise:
    raise Exception('InvalidDifficulty') if self.difficulty is not
        between 0 and 1.
```

- `_quitwindow()`

Destroy/quit the windows after using it.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `_getWordlist()`

Read the excel sheet (the local dictionary with frequencies of the words) and convert it into a dictinary with the key being the words and and the value being its `frequency/max_frequency`.

```
Args:
    None

Return:
    A dictionary

Raise:
    None
```

- `_getArticle(text)`

Read the text, cut it into words, then add them into a set and return.

```
Args:
    text: The article we want to anaylze.

Return:
    A set containing the words.

Raise:
    None
```

- `_dictLookup(word, d)`

Local wordlist lookup. Check if the word is in the wordlist and get its frequency. Also check the stripped words if the word matches any common suffixes.

```
Args:
    word: the word to look up
    d: The dictionary to look up

Return:
    If any result is found, then return a tuple of (word, frequency).
    If the word (or any of the stripped version) is not in the list, then
    return None.

Raise:
    None
```

- `_setLookup(s, dictionary)`

Look up each word in the set in the local wordlist. If the word is found and its frequency is below the self.difficulty, then recognize it as difficult word. Stopwords are removed.

```
Args:
    s: a set containing the words to be looked up.
    dictionary: the dictionary with the key be the words and the value be
    its frequency.

Return:
    a list containing the difficult words found in the local wordlist.

Raise:
    None
```

`cralwer.py`

This file defines a namedtuple Entry to represent a dictionary entry, and a LookupRequest class to represent a lookup in Cambridge Dictionary for each word. The required modules are:

- collections
- re (reqular expression)
- requests
- urllib
- bs4 (BeautifulSoup)

Below we list all the classes defined in this file.

Entry

A namedtuple representing an entry in a dictionary of a looked-up word.

Usage:

```
Entry(word, pos, pronunciation, listOfDefinitions, listOfExamples)
```

Attributes:

- word: a string saving the word.
- pos: part of speech. A string.

- pronounciation: A string, which is the pronunciation of that word.
- definitions: A list of definition of the word. Must have the same order with examples (see examples below).
- examples: A list of list of examples of the word, corresponding the definitions. Must have the same order with definitions (see examples below.)

Example:

```
w = Entry('dynamic',

    ['adjective. 思維活躍的；活潑的，充滿活力的，精力充沛的',
    'adjective. 不斷變化的；不斷發展的'],

    [['She's young and dynamic and will be a great addition to the team.',
    'We need a dynamic expansion of trade with other countries.'],
    ['Business innovation is a dynamic process.',
    'The situation is dynamic and may change at any time.']],

    '/daɪˈnæm.ɪk/')

w.word = 'dynamic'

w.definitions =
    ['adjective. 思維活躍的；活潑的，充滿活力的，精力充沛的',
    'adjective. 不斷變化的；不斷發展的']

w.examples =
    [['She's young and dynamic and will be a great addition to the team.',
    'We need a dynamic expansion of trade with other countries.'],
    ['Business innovation is a dynamic process.', 'The situation is
    dynamic and may change at any time.']]

w.pronunciation() = '/daɪˈnæm.ɪk/'
```

LookupRequest

A class to save a word, look it up and save its lookup results.

Usage

To construct an object, use:

```
w = LookupRequest('MyWord')
```

To tell it to look up itself, use

```
w.onlineLookup()
```

Finally, export the result, which is usually a list of Entry objects, with

```
result = w.export()
```

If the lookup failed, that is, no entry is found, then export will return `None`.

Attributes

- `_word`: A string. The word to be looked up (The original word inserted while constructing the object).
- `_entries`: A list of Entry objects. The entries found while looking up the word. Set to `None` before lookup.

Class Methods

- `__init__(word)`

Construct a LookupRequest instance with a word. Saves the word as an attribute without looking it up.

```
Args:
    word: the word to be looked up

Returns:
    None

Raises:
    None
```

- `onlineLookup()`

Packed lookup method. Deals exceptions and British spellings.

Calls `self._direcOnlineLookup()`, see its explanation below for detailed description. If the lookup result turns out to be a "...的美式拼寫", then lookup again with the british spelling. If NoEntryFound exception was raised, then set `self._entries` to be None.

```
Args:
    None

Returns:
    None

Raises:
    None
```

- `export()`

Export the list of entries found.

```
Args:
    None

Returns:
    A list of Entry objects.

Raises:
    None
```

- `_directOnlineLookup([target=None, replace=None])`

Look up the word and saves a list of Entry objects to `self._entries`.

Look up the word in Cambridge online dictionary and return a list of Entry objects.

```
Args:
    target: the word to look up. If the target is None, then lookup the
        word saved in self._word. Default value: None
    replace: Default value: None. If replace is not none, then the target
        word will be replaced with replace in the entries.

return:
    None

raises:
    Raises NoEntryFound exception if the page is empty.
```

`imgrecog.py`

This file can lookup words from an image. The user select a picture, then it will be shown in a window. When the user click on the word in the picture, this program will attempt to recognize the word and generate the vocabulary card. Note that the picture must be very well taken so that the lines are not tilted or bent too much. Some examples are provided in `/main/imgexample/`. The required modules are:

- PIL
- numpy
- pytesseract
- re
- tkinter

Below we list all classes we used in this file.

TextPicture

A picture containing text to be recognized.

Constants:

- `_BNW_THRESHOLD`

When the gray scale number of a pixel is smaller than this constant, it will be recognized as a white pixel and set to 0. Otherwise, it will be recognized as a black pixel and set to 1.

- `_LINE_THRESHOLD`

When the number of black pixels in a raw is less than this number, it will be recognized as a white line.

- `_SPACE_THRESHOLD`

When the black pixels in a column is less than this constant, it will be considered as a white column.

Attributes:

- `_originalImg`: an Image object which is a colored (possibly scaled) image chosen by the user.
- `_imgArray`: a 2D `np.array` object containing only `0` and `1`, representing a black-and-white version of the image.
- `_height`: the height of the (possibly scaled) image.
- `_width`: the width of the (possibly scaled) image.

- `_horizontalSum`: A 1D `np.array` containing the number of 1 of each row.

Class Methods:

- `_is_similar(s1,s2)`

Determine whether two words s1 and s2 are similar or not.

We would compare the length and the characters in them to determine whether they are similar or not.

```
Args:
    s1, s2: two strings to be compared.

Return:
        True: if they are similar.
        False: if they are not.

Raises:
    None
```

- `_extractLine(lineUpperBound, lineLowerBound)`

This method will analyze the a region which is defined by lineUpperBound and lineLowerBound. This region should contain the line to be analyzed.

We will first sum up vertically to detect the white column. If the number of black pixels in a column is less than `_SPACE_THRESHOLD`, we recognize it as a white column. Then, we find the wider white column to be the space between two words. Everything between two spaces is a word. Lastly, we return a list of indices to represent the coordinates of each word.

```
Args:
    lineUpperBound: The upper bound of raw of the interest region.
    lineLowerBound: The lower bound of raw of the interest region.

Return:
    wordIndices: a list of index of divide of words.

Raise:
    None
```

- `recognizeWord(wordX,wordY):`

  This function is designed to recognize the word on the position(wordX, wordY).

  Here, we will first find the nearst white raws to detect the line which the word belong to. Then, use `_extractLine(lineUpperBound, lineLowerBound)` to divide the words.

  Then, we put the image of the line into pytesseract to transform image to English. Use the order of the word in the string to get `targetWordFromLine`. But sometimes the order may be detected wrong. So we chop the image of the word and use pytesseract to get `targetWordFromWord`, which may have lower precision than the word detected in whole line.

  Last, we use `_is_similar(s1, s2)` to compare `targetWordFromLine` and `targetWordFromWord`. If they are similar, then return `targetWordFromLine`, which has higher precision. When they are not similar, it implies that the order of the word in line may be detected wrong. Thus, `targetWordFromLine` may be wrong, so we return `targetWordFromWord` instead.

    ```
    Args:
             WordX, WordY: The position of the word we want to recognize.

    Return:
            The word we recognize. Type: str

    Raise:
            None
    ```

- `_bindEvent(event)`:

When the event is occured, we will return the word on the position (event.x, event.y)

```
Args:
    event: The event that triggers the function recognizeWord.

Return:
    The word recognized at position (event.x, event.y)

Raise:
    None
```

ImgRecognitionWindow

This class defines the window showing the image. The user clicks on words they want to look up and the word will be listed in the wordlist on the right. This class inherit the class `TextPicture`.

Attributes

- `_img_path`: The path to the image to be analyzed.
- `_picWindow`: A `tk.Toplevel` object. The Window containing the picture.
- `_wordWindow`: A `tk.Toplevel` object. The window containing the word found in the image.
- `_wordlistFrame`: A `WordlistWindow` object. The frame in `_wordWindow` containing the words and the checkboxes.

Class Methods

- `__init__(self)`

This constructor will create two windows.

`_picwindow` will show the picture the user has chosen (We will let the user choose the file they want to be analyzed). Then, the user can click the word and the word will be recognized.

`_wordWindow` will show the selected words. The user can choose which words they want to make the vocabulary cards.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `_quitwindow()`

Destroy/quit the windows.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `bindEvent(event)`

When the event occured, We will detect the word users click and add it to the `_wordWindow`.

```
Args:
    None

Return:
    None

Raise:
    None
```

`main.py`

This file create a menu window for you to select what deck you want to add card in and what lookup function you want to use. It contains an option menu to select the deck, a button to refresh the decks, and three buttons to select the functions. The required modules are:

- tkinter
- messagebox (tkinter)
- filedialog (tkinter)
- abspath (os.path)
- dirname (os.path)
- chdir (os)

Below we list the class defined in this file.

lookupGUI

A class to create a menu window. It contains 4 buttons and an option menu.

`refreshButton` is bound with the function `_refreshDecks`. When users click on it, the function will update the connection with anki and refresh the decks listed in the option menu.

`word_lookup_button`, `article_lookup_button`, `image_lookup_button` are bound with the methods `_wordlookup`, `_articlelookup`, `_imagelookup`, respectively. The methods use the modules to create new windows.

`_decksmenu` is an option menu which lists all the decks of the user's anki. The users can select one of it to add new cards in. If the program can not connect with the user's anki, this menu will show nothing.

Attributes:

- `_master`: A `tk.Tk` object. The master of this program.
- `_deck_name_prompt`: A `tk.Label` object. The text to ask the user either to launch Anki or to choose a deck.
- `_targetDeck`: The deck name that all the cards will be added to.
- `_decknames`: A tuple containing all the decks in the user's Anki account.
- `_decksmenu`: A `tk.OptionMenu` object listing all the decknames.

Class Methods:

- `_imagelookup()`

When the user click on `image_lookup_button`, check if the user has selected the deck. If yes, open `ImgRecognitionWindow`. Otherwise, show a message box to remind the user.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `_wordlookup()`

When the user click on `word_lookup_button`, check if the user has selected the deck. If yes, open `WordLookupWindow`. Otherwise, show a message box to remind the user.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `_articlelookup()`

When the user click on `article_lookup_button`, check if the user has selected the deck. If yes, open `ArticleRecognitionWindow`. Otherwise, show a message box to remind the user.

```
Args:
    None

Return:
    None

Raise:
    None
```

- `_getDecks()`

Called by the method `_updateDeckNames`. This method use the API to get the deck names in the user's anki.

```
Args:
    None

Return:
    A list contains all the deck names in the user's anki.

Raise:
    None
```

- `_updateDeckNames()`

Called by the method `_refreshDecks`. this method changes the list got from `_getDecks` into tuple and put them in the attribute `_decknames`. If the program can not connect with the user's anki, it will show a warning caption.

```
Args:
    None

Return:
    None

Raise:
    Raise Exception('Failed to connect with API. Please check that you
    have all the prerequisite installed then click"refresh".') if the
    program can not connect with the anki API.
```

- `_refreshDecks()`

When the user click on `refreshButton`, reconnect to the user's anki and update the names of the deck in it. Also, refresh the options in the option menu.

```
Args:
    None

Return:
    None

 Raise:
    None
```

`wordlist_cls.py`

This file defines the WordlistWindow class, which is the window with all the word added to be looked up. It uses threading module to lookup in the background.

Below we list all the classes defined in this file.

WordlistWindow

A class defining the frame listing all the words to be added.
The design of this class is based on the answer here

This class inherit tk.Frame. Initiate the window by constructing an instance as a normal tk.Frame:

```
    w = WordlistWindow(master, quitFunc)
```

Then pack/grid to show the frame. Use

```
    w.newWord('MyWord')
```

to add a new word. The word should pop up in the list immediately, unless it is already in the list. A lookup process will be initiated in the background once the object is constructed. Each word added with `newWord()` method will be added to a queue, and will be looked up one by one in the background with threading module. The words added to the list should have its checkbutton checked by default. The user can click on the "done" button to quit and `quitFunc()` will be called, which can be used to close the window.

Attributes

- `vscrollbar`: The vertical tk.Scrollbar object on the right.
- `canvas`: The `tk.Canvas` in the background of the frame.
- `interior`: `a tk.Frame` object that everything lie on.
- `_status`: a `tk.Label` object showing the length of the queue.
- `_quitButton`: a `tk.Button` object that will quit the window while pressed.
- `_cbvar`: A list of `tk.BooleanVar` objects saving and monitoring the value of the checkbutton of each word.
- `_cb`: A list of `tk.Checkbutton` objects, one for each word, and linked to the corresponding `tk.BooleanVar` objects in `self._cbvar`.
- `_queue`: A list of strings saving the words that are added via `self.newWord(word)` and are not yet looked up.
- `_finished`: A list of list of Entry objects saving the lookup result of each word. `None` if the lookup failed.
- `_finishedWord`: A list of strings saving the words that have been looked up.
- `_thread`: A boolean. If set to `False`, the lookup threading will pause.
- `_interval`: The length of time to wait while the queue is empty.
- `_threadInstance`: A `threading.Thread` object controlling the lookup in the background.
- `_quitFunction`: A function. Will be called when `self._quitButton` is pressed to kill the window.

Class Methods

- `__init__(master, quitFunc [, **kwargs])`

Construct a modified tk.Frame object with scrollbar and word checklist.

Constructor. Inherit the Frame class from tk, while adding a scrollbar, and word listing feature. Takes all arguments as tk.Frame.

```
Args:
    same as tk.Frame.

Return:
    None

Raises:
    None
```

- `newWord(word)`

Add new words

function to add new word. load the word in to the queue, then the method `_lookupThreading` will look them up in the background.

```
Args:
    word: The word to be added.

Returns:
    None

Raises:
    None
```

- `_updateStatus()`

Update the status label to indicate the queue length.

- `_lookupThreading()`

Look up words in `self._queue` in the background.

Uses threading module to implement multitasking, so it will continue to lookup words (the speed depends on the internet speed) while the user input words (the speed depends on CPU and GPU).

```
Args:
    None

Returns:
    None

Raises:
    None
```

- `_quitAndAdd()`

Add the words checked and quit.

Called when the user hit the "quit" button. The function wait until all cards have been looked up (the queue is empty), then add all cards to deck, and finally quit the window.

```
Args:
    None

Returns:
    None

Raises:
    None
```

- `_set_scrollregion([event=None])`

Update scroll region of the scrollbar.

`word_lookup.py`

This file creates a lookup window that you can key in the words you want to search. Whenever you press "Enter", it will catch the word you just keyed in. Then you can choose what words you want to make the vocabulary cards in anki. The required module is:

- tkinter

Below we list the class defined in this file.

WordLookupWindow

Create two windows.

One is `_inputWindow`, which shows a text box for users to key in words.

The other is `_wordlistWindow`, which shows the words the users just keyed in.

Attributes:

- `_wordlistWindow`: A `tk.Toplevel` object. The window showing the wordlist.
- `_inputWindow`: A `tk.Toplevel` object. The window with the textbox that allow the user to input words.
- `_inputBox`: A `tk.Text` object. The textbox that allow the user to input words.
- `_wordlistFrame`: A `WordlistWindow` object. The frame with the words that the user have typed and checkbuttons for each word.

Class Methods:

- `_quitWindow()`

Destroy/quit the windows after users click "Done".

```
Args:
    None

Return:
    None

Raise:
    None
```

- `_fireOnEnter(event)`

When the users press "Enter", catch the last word the users just keyedin and put it in the wordlist window.

```
Args:
    event: The action users did. We bind the "Enter" button to this
    function, so there is no need to fill this argument.

Return:
    None

Raise:
    None
```

Collaborators

- 張家翔 (hsiang20): API adaptor (`add_card.py`), single word lookup feature GUI design (`word_lookup.py`), main menu GUI design (`main.py`).
- 徐鼎翔 (AlbertHsuNTUphys): Image recognition feature (`imgrecog`), article lookup GUI design (`article_lookup.py`).
- 王昊謙 (fhcwcsy): Wordlist window GUI design (`wordlist_cls`), crawler (`crawler.py`), speed enhancement in image recognition, article lookup feature (my hw2), final modification in all files.