pytabby

Release 0.1.0

David Taylor

Mar 14, 2020

CONTENTS

A *flexible, non-opinionated*, **tabbed** menu system to interactively control program flow for terminal-based programs. It's a class with one sole public method which runs in a `while` loop as you switch tabs (if you want tabs, that is; you're free not to have any) or if you enter invalid input, and then returns a string based on the value you selected that you can use to control the outer program flow.

Of course, you can run the class itself in a `while` loop in the enclosing program, getting menu choice after menu choice returned as you navigate a program.

Blog post about why I did this.

CHAPTER

ONE

INSTALLATION

```
pip install pytabby
```

Meow.

CHAPTER

TWO

DOCUMENTATION

On readthedocs.

CHAPTER

THREE

USAGE

```
from pytabby import Menu
myconfig = Menu.safe_read_yaml('path/to/yaml')
# or Menu.read_json() or just pass a dict in the next step
mymenu = Menu(myconfig)
result = mymenu.run()

if result == 'result1':
    do_this_interesting_thing()
elif result == 'result2':
    do_this_other_thing()
# etc...
```

CHAPTER

FOUR

SEE IT IN ACTION!

CHAPTER

FIVE

FAQ

***Why did you make this?** Well, it was one of those typical GitHub/PyPI scenarios, I wanted a specific thing, so I made a specific thing and then I took >10X the time making it a project so that others can use the thing; maybe some people will find it useful, maybe not. I like running programs in the terminal, and this allowed me to put a bunch of utilities like duplicate file finders and bulk file renamers all under one umbrella. If you prefer GUIs, there are plenty of simple wrappers out there,

Why can't I return handlers? Out of scope for this project at this time, but it's on the Wish List. For now, the Menu instance just returns strings which the outer closure can then use to control program flow, including defining handlers using control flow/if statement based on the string returned by Menu.run().

Why are my return values coming in/out strings? To keep things simple, all input and output (return) values are converted to string. So if you have `config['tabs'][0]['items][0]['item_returns'] = 1`, the return value will be '1'.

Why do 'items' have both 'item_choice_displayed' and 'item_inputs' keys? To keep things flexible, you don't have to display exactly what you'll accept as input. For example, you could display 'yes/no' as the suggested answers to a yes or no question, but actually accept ['y', 'n', 'yes', 'no'], etc.

I have 'case_sensitive' = False, but my return value is still uppercase. `case_sensitive` only affects inputs, not outputs

What's up with passing a dict with the tab name as a message to Menu.run()? The message might be different depending on the tab, and `run()` only exits when it returns a value when given a valid item input. It changes tabs in a loop, keeping that implementation detail abstracted away from the user, as is right.

CHAPTER

SIX

DEPENDENCIES

- `PyYAML>=5.1`
- `schema>=0.7.0`

Note: You can have two or more tabs, in which case you will see the tab headers you can switch between above the menu choices, or you can

CHAPTER

SEVEN

CONTENTS

7.1 Tutorial

Everything depends on a good, valid config file. For this tutorial, we'll use the following short YAML (found in the Github repo as `example_configs/tutorial.yaml`):

```
case_sensitive: False
screen_width: 80
tabs:
  - tab_header_input: a
    tab_header_description: Menu a
    items:
      - item_choice_displayed: 1
        item_description: First choice
        item_inputs:
          - 1
        item_returns: choice1
      - item_choice_displayed: 2
        item_description: Second choice
        item_inputs:
          - 2
        item_returns: choice2
  - tab_header_input: b
    tab_header_long_description: You have just selected Menu B
    items:
      - item_choice_displayed: x,y,x
        item_description: x or y or z
        item_inputs:
          - x
          - y
          - z
        item_returns: xyz
      - item_choice_displayed: q
        item_description: Quit
        item_inputs:
          - Q
          - qUiT
          - bye
        item_returns: quit
```

This is a config for two tabs, each with two choices.

First, import the `Menu` class:

```
>>> from pytabby import Menu
```

Then, you need a config file. There are two static methods you can call from the uninstantiated class, `safe_read_yaml(path)` and `read_json` path. Or you can write a python dict and pass that. Here, let's read the YAML file.

```
>>> config = Menu.safe_read_yaml('tutorial.yaml')  # the file shown above
```

And now instantiate the class with the config dict:

```
>>> menu = Menu(config)
```

At this point, if your config has an invalid schema, the Menu class will raise an InvalidSchemaError and will output **ALL** the schema violations to stderr. (As opposed to raising just one, Then you fixing it, then raising another, you fixing it, etc.)

If the menu instance is instantiated without errors, you can just run it!

```
>>> menu.run()
```

You'll see the following printed to stdout:

```
[a:Menu a|b]
 ======== --
[1] First choice
[2] Second choice
?:
```

Note that the first tab, a, is underlined, showing that it's the active tab. Tab b has no description in the config, so it's very short.

Now enter 'c' (an invalid choice) at the prompt. A new line appears (the program does not resend the entire meny to stdout):

```
?: c
Invalid, try again:
```

Now enter 'b' to switch to that tab. The following is sent to stdout:

```
Invalid, try again: b
Change tab to b
You have just selected Menu B

[a:Menu a|b]
 -------- ==
[x,y,x] x or y or z
[q    ] Quit
?:
```

As you can see, the second tab, `b`, is now underlined. (It really is more obvious if you use descriptions.)

The program output `Change tab to` & the tab_header_input

Since for this second tab, 'tab_long_description' was defined, that was printed as well (`You have just selected Menu B`, how boring).

Now let's actually submit a valid choice (an invalid choice will give the same 'Invalid, try again: ' message as above).

```
?: x

('b', 'xyz')
```

Now the menu has returned a value, a tuple of the tab input and the return value. The tuple is returned because different tabs could have the same return value. If there were only one tab, only the return value `'xyz'` would have been returned.

7.2 API

class pytabby.**Menu**(*config*, *start_tab_number=0*)

Base class to import to create a menu

Parameters

- **config** (*dict*) – a nested dict, in a schema which will be validated, containing everything needed to instantiate the Menu class
- **start_tab_number** (*int*) – default 0, the number of the tab to start at

safe_read_yaml(*path_to_yaml*)

static method to read a yaml file into a config dict

read_json(*path_to_json*)

static method to read a json file into a config dict

run(*message=None*)

Displays menu at currently selected tab, asks for user input and returns it as a string

Examples

```
>>> config = Menu.safe_read_yaml('config.yaml')
>>> menu = Menu(config, start_tab_number=0)
>>> result = menu.run()
>>> if result = "action name":
>>>         my_great_function()
```

```
>>> # with a submenu
>>> config = Menu.safe_read_yaml('config.yaml')
>>> menu = Menu(config, start_tab_number=0)
>>> result = menu.run()
>>> if result = "further options":
>>>     submenu = Menu(submenu_config)
>>>     if submenu_result = "action name":
>>>         my_great_function()
```

7.3 Annotated config file

Here is a YAML config file, annotated to explain all the keys/values/lists etc.

```yaml
case_sensitive: False  # optional, boolean, default False
screen_width: 80  # optional, integer, default 80
tabs:  # if there's only one tab, there should be no headers. In fact, you can leave␣
↪out the tabs
       # key entirely and have 'items' as a top-level key.
  - tab_header_input: a  # this is the input that will change to this header; it can␣
↪be any length
    tab_header_description: First tab  # this key is optional, or it can be None. It␣
↪is displayed next to the tab_header_input
    tab_header_long_description: Changing to First tab  # this key is optional, or it␣
↪can be set to None.
                                                        # it is displayed only when␣
↪changing tabs
    items:
      - item_choice_displayed: x  # an element to be put within square brackets to␣
↪the left of every choice/item line
                                  # note that it does not necessarily have to␣
↪correspond at all to item_inputs, although
                                  # it probably should
        item_description: Choice x  # displayed to the right of item_choice_displayed
        item_inputs:  # not displayed, a list of all inputs that will trigger the␣
↪return of this item's 'item_returns'
          - x
        item_returns: xmarksthespot  # just a string
      - item_choice_displayed: y,z  # note I put a comma here for clarity, but this␣
↪field is not parsed, I could have put
                                    # anything eg "yz", "y|z", "y or z", "fizzbin",␣
↪etc.
        item_description: Choice y or z
        item_inputs:
          - y
          - z
        item_returns: yorz
  - tab_header_input: bee  # the second tab, note we can require a multi-letter input.
↪ This one has no descriptions.
    items:
      - item_choice_displayed: z  # this overlaps with the other tab, but that's okay,
↪ it's a different tab
        item_description: Choice z
        item_inputs:
          - z
        item_returns: z
      - item_choice_displayed: spam  # again multi-letter
        item_description: Surprise!
        item_inputs:
          - ham
          - jam
          - lamb  # none of these match the choice displayed! Bad idea, but they don
↪'t have to!
        item_returns: 1001010001010  # why not?
```

7.4 Example app

We're going to use `pytabby` to control program flow in an app that takes a directory full of files and allows you to categorize them into subdirectories.

There are two menus: [d]irectory management, where we see if folders exist and create them if needed; and [f]ile management, where we assign files to subfolders.

Our subfolders will be named `interesting` and `boring`.

Here's the python file, `app.py`. (It can also be found in the project github repo under example_app/). Instead of dealing with an external YAML config file, I've just hardcoded the config as a dict into the python app:

```
"""A simple app that shows some capabilities of pytabby."""

import glob
import os
import shutil

from pytabby import Menu

# to make this a self-contained app, hardcode the config dict

CONFIG = {
    "case_sensitive": False,
    "screen_width": 80,
    "tabs": [{"tab_header_input": "subdirs",
              "items": [{"item_choice_displayed": "c",
                         "item_description": "Create missing subdirectories",
                         "item_inputs": ["c"],
                         "item_returns": "create_subdirs"},
                        {"item_choice_displayed": "h",
                         "item_description": "Help",
                         "item_inputs": ["h"],
                         "item_returns": "help"},
                        {"item_choice_displayed": "q",
                         "item_description": "Quit",
                         "item_inputs": ["q"],
                         "item_returns": "quit"}]},
             {"tab_header_input": "files",
              "items": [{"item_choice_displayed": "i",
                         "item_description": "Move to interesting/",
                         "item_inputs": ["i"],
                         "item_returns": "interesting"},
                        {"item_choice_displayed": "b",
                         "item_description": "Move to boring/",
                         "item_inputs": ["b"],
                         "item_returns": "boring"},
                        {"item_choice_displayed": "s",
                         "item_description": "Skip",
                         "item_inputs": ["s"],
                         "item_returns": "skip"}]}]}


def print_help():
    """Print help string to stdout"""
    help_text = (
        "This app goes through the contents of a directory and allows you to␣
↪categorize the files, "
```

(continues on next page)

(continued from previous page)

```
        "either moving them to subdirectories called interesting/ and boring/ or␣
↪skipping them. This "
        "functionality is handled by the second tab\n    The first tab allows you to␣
↪check if the "
        "subdirectories already exist, allows you to create them if they are missing,␣
↪shows this help "
        "text and allows you to quit the app\n"
    )
    print(help_text)


def get_directory():
    """Get the name of a directory to use, or uses the current one"""
    valid = False
    while not valid:
        directory = input("Enter directory (blank for current): ")
        if directory.strip() == "":
            directory = os.getcwd()
        if os.path.isdir(directory):
            valid = True
        else:
            print("That directory does not exist.")
    return directory


def get_files():
    """Determine sorted list of files in the current working directory"""
    files = []
    for item in glob.glob("./*"):
        # add current .py file in case it's in the directory
        if os.path.isfile(item) and os.path.split(item)[1] != os.path.split(__file__
↪)[1]:
            files.append(item)
    return sorted(files)


def create_subdirectories():
    """Create subdirectories if they do not exist"""
    for subdir in ["interesting", "boring"]:
        if os.path.isdir(subdir):
            print("./{0}/ EXISTS".format(subdir))
        else:
            os.mkdir(subdir)
            print("./{0}/ CREATED".format(subdir))
    print("")


def move_to_subdir(filename, subdirname):
    """Move filename to subdirname"""
    if os.path.isfile(os.path.join(subdirname, filename)):
        raise ValueError("File already exists in that subdirectory!")
    shutil.move(filename, subdirname)
    print("{0} moved to ./{1}/".format(filename, subdirname))
    print("")

def main_loop():  # noqa: C901
    """Contain all the logic for the app"""
```

(continues on next page)

(continued from previous page)

```
    menu = Menu(CONFIG)
    files = get_files()
    current_position = 0
    quit_early = False
    files_exhausted = False
    while not (quit_early or files_exhausted):
        filename = files[current_position]
        files_message = "Current file: {0} of {1}: {2}".format(current_position + 1,␣
↪len(files),
                                                               os.path.
↪split(filename)[1])
        # message will be shown only when we are on the files tab
        result = menu.run(message={'files': files_message})
        if result == ("subdirs", "create_subdirs"):
            create_subdirectories()
        elif result == ("subdirs", "help"):
            print_help()
        elif result == ("subdirs", "quit"):
            quit_early = True
        elif result[0] == "files" and result[1] in ["interesting", "boring"]:
            if not os.path.isdir(result[1]):
                raise ValueError("Directory must be created first")
            move_to_subdir(files[current_position], result[1])
            print('File moved to {}'.format(result[1]))
            current_position += 1
            files_exhausted = current_position >= len(files)
        elif result == ("files", "skip"):
            current_position += 1
            files_exhausted = current_position >= len(files)
        else:
            raise AssertionError("Unrecognized input, this should have been caught by␣
↪Menu validator")
    if files_exhausted:
        print("All files done.")
    else:
        print("Program quit early.")


if __name__ == "__main__":

    CWD = os.getcwd()
    os.chdir(get_directory())
    main_loop()
    os.chdir(CWD)
```

You can try this out on any folder of files; in the GitHub repo, there's a folder called example_app with this script, `app.py`, and six photos downloaded from Unsplash, and resized to take up less space. Note that this program doesn't *show* the images, but feel free to build that capability into it!

Here's an example terminal session using the above script:

```
example_app$ ls
app.py                              cade-roberts-769333-unsplash.jpg
prince-akachi-728006-unsplash.jpg    tyler-nix-597157-unsplash.jpg
brandon-nelson-667507-unsplash.jpg  colton-duke-732468-unsplash.jpg
raj-eiamworakul-514562-unsplash.jpg
```

There are six jpgs we will classify as interesting or boring, plus the app.py script that is smart enough to ignore itself when moving files. The `boring` and `interesting` folders are not yet present.

```
example_app$ python app.py
Enter directory (blank for current):

[subdirs|files]
 ======= ------
[c] Create missing subdirectories
[h] Help
[q] Quit
?: c
./interesting/ CREATED
./boring/ CREATED
```

If we try to create the directories again, we'll just be told they already exist

```
[subdirs|files]
 ======= ------
[c] Create missing subdirectories
[h] Help
[q] Quit
?: c
./interesting/ EXISTS
./boring/ EXISTS


[subdirs|files]
 ======= ------
[c] Create missing subdirectories
[h] Help
[q] Quit
?: h
This app goes through the contents of a directory and allows you to
categorize the files, either moving them to subdirectories called
interesting/ and boring/ or skipping them. This functionality is
handled by the second tab
    The first tab allows you to check if the subdirectories already
exist, allows you to create them if they are missing, shows this help
text and allows you to quit the app


[subdirs|files]
 ======= ------
[c] Create missing subdirectories
[h] Help
[q] Quit
?: files
Change tab to files

[subdirs|files]
 ------- ======
[i] Move to interesting/
[b] Move to boring/
[s] Skip
Current_file: 1 of 6: brandon-nelson-667507-unsplash.jpg
?: i
./brandon-nelson-667507-unsplash.jpg moved to ./interesting/
```

(continues on next page)

(continued from previous page)

```
File moved to interesting

[subdirs|files]
 ------- ======
[i] Move to interesting/
[b] Move to boring/
[s] Skip
Current_file: 2 of 6: cade-roberts-769333-unsplash.jpg
?: b
./cade-roberts-769333-unsplash.jpg moved to ./boring/

File moved to boring

[subdirs|files]
 ------- ======
[i] Move to interesting/
[b] Move to boring/
[s] Skip
Current_file: 3 of 6: colton-duke-732468-unsplash.jpg
?: s

[subdirs|files]
 ------- ======
[i] Move to interesting/
[b] Move to boring/
[s] Skip
Current_file: 4 of 6: prince-akachi-728006-unsplash.jpg
?: i
./prince-akachi-728006-unsplash.jpg moved to ./interesting/

File moved to interesting

[subdirs|files]
 ------- ======
[i] Move to interesting/
[b] Move to boring/
[s] Skip
Current_file: 5 of 6: raj-eiamworakul-514562-unsplash.jpg
?: i
./raj-eiamworakul-514562-unsplash.jpg moved to ./interesting/

File moved to interesting

[subdirs|files]
 ------- ======
[i] Move to interesting/
[b] Move to boring/
[s] Skip
Current_file: 6 of 6: tyler-nix-597157-unsplash.jpg
?: i
./tyler-nix-597157-unsplash.jpg moved to ./interesting/

File moved to interesting
All files done.
```

Now the program exits, and we can verify all the files are where we expect

```
example_app$ ls
app.py  boring  colton-duke-732468-unsplash.jpg  interesting
example_app$ ls boring/
cade-roberts-769333-unsplash.jpg
example_app$ ls interesting/
brandon-nelson-667507-unsplash.jpg  prince-akachi-728006-unsplash.jpg
raj-eiamworakul-514562-unsplash.jpg  tyler-nix-597157-unsplash.jpg
```

7.5 Sample InvalidSchemaError output

To give you an idea of the different error messages (all output at once under one umbrella exception), let's use the following YAML with many problems, indicated by comments:

```yaml
case_sensitive: 'This should be a boolean'
screen_width: 'This should be an integer'
tabs:
  - # there is no tab_header_input for this tab
    # the other two keys missing is ok, they're optional
    items:
      - item_choice_displayed: choice1
        item_description: descript1
        item_inputs:
          - # item inputs is an empty list
        item_returns: return1
  - tab_header_input: a  # this will conflict with an item input in another tab
    items:
      - item_choice_displayed: 1
        item_description: descript1
        item_inputs:
          - 1  # int is okay, but it will conflict with another input in the same list
        item_returns: None  # must be coerceable to string, can't be None
      - item_choice_displayed: 2
        item_description: descript2
        item_inputs:
          - 1  # conflicts with previous item
        item_returns: 1
  - tab_header_input: b
    items:
      - item_choice_displayed: a
        item_description: descripta
        item_inputs:
          - a  # conflicts with the input to switch to another tab
        item_returns: a
      - item_choice_displayed: foo
        item_description: bar
        item_inputs:
          - spam  # conflicts with previous item
        # missing item_returns
```

And here's what happens when we try to initialize a Menu instance with that config:

```
>>> from pytabby import Menu
>>> config = Menu.safe_read_yaml('sampleerror.yaml')
>>> menu = Menu(config)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
  File "<<path redacted>>pytabby/menu.py", line 58, in __init__
    validators.validate_all(self._config)
  File "<<path redacted>>pytabby/validators.py", line 388, in validate_all
    raise InvalidInputError("\n".join(printed_message))
pytabby.validators.InvalidInputError:
Errors:
1. schema.SchemaError: Key 'case_sensitive' error: 'This should be a boolean' should␣
↪be instance of 'bool'
2. tab#0: schema.SchemaMissingKeyError: Missing key: 'tab_header_input'
3. tab#0,item#0,valid_entry#0: schema.SchemaError: <lambda>(None) should evaluate to␣
↪True
4. tab#2,item#1: schema.SchemaMissingKeyError: Missing key: 'item_returns'
5. In tab#1, there are repeated input values including tab selectors: [(1, 2)].
6. In tab#2, there are repeated input values including tab selectors: [('a', 2)].
>>>
```

As you can see, instead of having to fix the errors one by one, the error message output six errors to fix at once.

It's a little unpythonic, but foolish consistency is the hobgoblin of bad design, to paraphrase.

Now, it's possible certain kinds of errors can be 'swallowed' by others, so there's no guarantee that you won't have to do more than one round of fixes in particularly problematic configs, but this should save you much more than half your time.

7.6 Blank config templates

For your convenience, here are blank config templates in .yaml, .json and .py (dict) formats. Add tabs and items as desired.

Remember, having only one tab is the same as having no tabs (there's nothing else to switch to), so in that case the 'tabs' key is optional, you can just have the 'items' key in the top level. And if you do have a 'tabs' key with only one items, it can't have 'tab_header_*' keys, because they would be meaningless and the program schema validator thinks you made a mistake by leaving out the other tabs.

```yaml
case_sensitive: False
screen_width: 80
tabs:
  - tab_header_input: tabinput1
    tab_header_description: tabdescript1
    tab_header_long_description: tablongdescript1
    items:
      - item_choice_displayed: choice1
        item_description: descript1
        item_inputs:
          - input1
        item_returns: return1
```

```json
{
    "case_sensitive": false,
    "screen_width": 80,
    "tabs": [
        {
            "tab_header_input": "tabinput1",
            "tab_header_description": "tabdescript1",
```

(continues on next page)

(continued from previous page)

```
            "tab_header_long_description": "tablongdescript1",
            "items": [
                {
                    "item_choice_displayed": "choice1",
                    "item_description": "descript1",
                    "item_inputs": [
                        "input1"
                    ],
                    "item_returns": "return1"
                }
            ]
        }
    ]
}
```

```
config = {'case_sensitive': False,
          'screen_width': 80,
          'tabs': [{'tab_header_input': 'tabinput1',
                    'tab_header_description': 'tabdescript1',
                    'tab_header_long_description': 'tablongdescript2',
                    'items': [{'item_choice_displayed': 'choice1',
                               'item_description': 'descript1',
                               'item_inputs': ['input1'],
                               "item_returns": 'return1' }
                    ]}]
         }
```

7.7 Changelog

This document records all notable changes to pytabby. This project adheres to Semantic Versioning.

7.7.1 0.1.0

- 2019-04-25: Initial release

7.8 Wish list

- a way to dynamically silence ("grey out", if this were a GUI menu system) certain menu items, which may be desired during program flow, probably by passing a list of silenced tab names and return values
- have an option to accept single keypresses instead of multiple keys and ENTER with the input() function, using `msvcrt` package in Windows or `tty` and `termios` in Mac/Linux. (This will make coverage platform-dependent, so it will have to be cumulative on travis and appveyor)
- incorporate ansimarkup (https://pypi.org/project/ansimarkup/) – is it cross compatible? Will it work with cmder.exe on windows? So the app could have really cool colored tabs!!! Would colorama work?
- Add MacOS CI (Circle?)
- Fix tests by (1) changing lazy regression tests to true tests, (2) relying less on ordering to be true unit tests, (3) monkeypatch inputs instead of that ugly menu.Menu._testing hacks

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

M

R

S