# MENU

A smart interface for TUI Python applications
alternative for argparse library

# Index

# 1 Functionality and advantages over argparse

## 1.1 Advantages over argparse library

1. Alternative for argparse
2. Allows user to play his Python script both: with arguments and as menu list
3. Additional functionalities
4. Ready interface for any prompt application.
5. Easy way of adding description / help

## 1.2 Functionalities

1. Check existence and correctness of added arguments (if they exist, need additional argument)
2. Check existence of sub-arguments (if given sub-argument exist on sub-argument list)
3. Show description / help
4. Print arguments as menu with possibility to get back to main menu
5. Add subargument to submenu list from within any function you are developing.

> **Tip.**
> You may check what functionalities Menu is giving you by simply copying whole application into one place and running main_file_example.py file. You can play around with run arguments to understand advantages of application.

# 2 Basic Technical Information

Menu is written in Python, independent from operational system – should work with any operational system that have installed Python. Menu use function exit from sys – standard Python library.

Code logic
Menu logic is contained within class 'Menu' which takes object as argument.
Class id divided between 6 main layers some of them have several functions, some of them only one function:
Presentation – contains functions for presentation

## 2.1 Work algorithm

1. Script gather 2 arguments (they are not necessary): `argument 1` and `argument 2`
2. Script check if gathered argument(s) are existing on argument / subargument list and if they are need.
3. Script may print help or lunch menu
4. Script may print submenu if argument have defined several subarguments
5. When arguments / subarguments are correct script lunch function assigned to argument and eventually pass subargument (argument 2) to assigned function.
6. Script ends application or go back to menu or submenu.

# 3 System Architecture Model

## 3.1 Files logic

`/menu`– all menu files

`main_file_example.py` – this is example file that purpose is to show you full potential of menu.

`menu.py` – logical application

`text.py` – file with all text for communication with user. All stored in one place to ease translation into any language.

`empty_pattern.py` – interface for your prompt application. Here where you start to writing code.

## 3.2 Script logic

Script is run by any other writing script that have been configured for the script / application. Your application, that you want to create have already framework / skeleton, that is `empty_pattern.py` file. File name as well as YourClassName class may be changed whatever name you want it to give. class MenuConfig stores all necessary (all that are existing) variables, data contained in this variables are used by menu script. Each time your script is runnig, it creates new object MenuConfig that stores all variables. Then it pass data to your Main Class and here it creates Menu object and again pass here MenuConfig instance variable, so Menu now can use all necessary data. Menu script checks if written by user arguments during lunch of your application are correct and act accordingly to configuration.

## 3.3 MenuConfig class – all config variables to set your application functions

All need config for your application you will find in class MenuConfig. Here you have variables which stands for your application name, type, description for help, menu arguments and other variables that will be used by script – and checking if argument given by user is correct. Some of the variables may not be clear at the beginning, so therefore each of them was explained in manual.

# 4 Manual

Manual is divided between 3 smaller parts. First tells you how to import Menu script for your location from different location, Second is dedicated to explain all basic config variables and Third is to explain how to use additional methods of Menu script, which may ease your life when writing application.

## 4.1 Importing menu script from specific localization (from path)

To use script / library you can just simply copy all files to chosen by you location and write you logical application in `empty_pattern.py` file. However, menu was written with idea to be universal, so you can use it in any python TUI (Text-based User Interface) application that you want to write.
So better way is to put menu script / library in some localization and import menu to any file you want to code in.
File main.py you can treat as pattern or interface for your application. It is better to use it, because it is general interface of your application.
If you want to use menu library in many locations, perform below steps:

1. put menu.py and menu_text.py file into chosen location:

   ```
   /chosen/by/you/location/menu/
   ```

2. make sure that files have all necessary access rights
3. within main.py file, which is interface pattern of your script, unhash lines 8-11:

   ```
   path_menu = '/chosen/by/you/location/menu/menu.py'
   path_menu_text = '/chosen/by/you/location/menu/menu_text.py'
   sys.path.append(os.path.dirname(os.path.expanduser(path_menu)))
   sys.path.append(os.path.dirname(os.path.expanduser(path_menu_text)))
   ```

Where:
`path_menu` and `path_menu_text` variables should be fulfilled with chosen by you location of `menu.py` and `menu_text.py` files.

## 4.2 MenuConfig class – all config variables to set your application functions

All variables will be here shown with default written value. All variables, or data structures are universal Python data structures. Used data structures are: variables (type: Strings and bools), lists and dictionaries.
All variables are stored within MenuConfig class, and are loaded when object of this class is created.
List of variables with default value and description:

`application_name = 'NAME'` – variable stores name of your application. Variable has to be String.

`application_type = 'SCRIPT / APPLICATION'` – variable stores info about type of your application. Regards what you put here, it only will be printed. So you may left here even empty String. Value has to be String.

`additional_application_description = 'DESCRIPTION'` – variable stores description about your application, what it does. You may put here long description in many sentences spreaded over several lines.

`need_second_argument = False` – variable take as argument `False` or `True`. Set it to `True`, only if you want that your application to demand from user always to give 2 arguments (always additional arguments)

`letter_for_menu = 'm'` – variable stores letter, that will recognize if user want to lunch menu, or want go to main menu from submenu argument list. Variable should not be changed.

`need_first_argument = True` – variable take as argument `False` or `True`. Set it to `True`, only if you want that your application to demand from user always to give additional argument for run specific. Set it to `False` when you write application that may be run without arguments.

`argument_list = { […] }` – argument list is a dictionary. Each element contain letter (argument) that will lunch specific function and description of functionality of this function. It is important to know, that argument is used in 2 phases:

1. Check if given by user argument exist in argument list
2. Print list of arguments with description in menu, or after help lunched

Example of list with all necessary values + one added function:

```
argument_list = {
                  'm': 'menu',
                  'h': 'help',
                  'q': 'quit (only for menu)',
                  'a': 'will lunch added function',
                  }
```

**WARNING.**

Please be aware that 3 first elements on list: menu, help, quit – this lines should not be deleted and letters of this options should not be changed. Otherwise script may work wrong, or throw an error.

`additional_argument_list = [ ]` – this list contains list of arguments for which you want to be check if argument was passed by user with subargument. For example, if you written `False` to variable `need_second_argument`, then you do not want every argument to demand additional argument 2 from user. But for some of arguments you may want to check if additional argument has been pass, because it may be necessary for some reason. In this list you can write all arguments that need additional argument. So for example if your application has argument `-a` and function underlying under this argument need second argument, just add it to `additional_argument_list` list. After running your application with `-a` argument, menu will check if second argument were given, and if not, will return information to person who lunched script, that second argument is need for `-a` argument.

Example of fulfilled list:

```
additional_argument_list = [ 'a', 'b', 'j' ]
```

**WARNING.**

Arguments on `additional_argument_list` list always have to be also existing on `argument_list` list.

`any_argument_have_submenu = False` – variable take as argument `False` or `True`. Set it to `True` only if next dictionary you want to fulfill. Set to `False` if you will not add any record to `submenu_arguments_list` - next dictionary.

`submenu_arguments_list = { }` – all existing here arguments have to existing on `argument_list` list. This is dictionary, where you can put subarguments list of one argument from `argument_list` list.

Example of usage:

Let's assume that you have fulfilled dictionary this way:

```
submenu_arguments_list =      {
      'c': [ 'sub-argument 1', 'sub-argument 2', 'sub-argument 3' ],
                              }
```

If you has fulfilled dictionary as above, then:

- when user lunch your application with `-c` option from prompt, will also have to give as second argument one of the value from submenu list related to `-c` argument, otherwise menu will return information that there is no such subargument as given, and will print all subarguments
- when user lunch `-c` option from menu, then menu will print submenu list as menu, and give possibility to choose one of submenu option, or back to main menu.

## 4.3 Additional methods – how to use them

Menu have 2 useful functions written to ease you work. Below this functions are described.

### 4.3.1 function goto_exit_or_main_menu()

Purpose of this function is break currently running function, and based on information how user did started function. If user:

- start with parameter from prompt – exit the script
- start function from menu – go back to main menu.

Regardless if started from menu or as argument, will return to user information stored within function argument.

Example of function usage:

```
self.menu_object.goto_exit_or_main_menu( 'value' )
```

Function always should be used in upper form. Only thing that should be changed in function is passed argument (In above example this is `'value')` text. Argument always should be String that contains information why function has been broken.

### 4.3.2 add_value_to_submenu_arguments ()

Purpose of the function is to add to any argument, that have list of subargumens, new subargument.

Explanation Example 1:

Imagine that you are writing application that is listing file from some directory once daily. Let's assume this option is lunched from `-a` option. It is done automatically by schedulder.

User that is using your application do not have access to whole your script directly. He only know how to use option `-b`, which stands for function that is printing for this user listed in `-a` option list of files. But this lists are stored for many days. Here it helps function. You will use it in function from option `-a` for `-b` subargument list.

It will automatically add desired value (it may be date) for `-b` subargument list.

Explanation Example 2:

Imagine that you have script that may be executed by many users. Script is dedicated to check group of reports. Any user may want to have his personal list of reports to check, important for himself. Also there may be list of general reports for everyone, or maybe for some reason one user likes personal list of reports created by someone else. Each list of reports to check, regardless if personal or general is stored within the text file.

Imagine all text personal files are stored in `/reports` location. Each user can create new text file and add it to `/reports` location by using `-a` function of your script.

Each of text files can be run (load to check) as subargument of `-b` argument. Now anytime any user that want to add new text file, he would also have to add this text file as subargument to `-b` to `submenu_arguments_list` in `MenuConfig` class. But function `add_value_to_submenu_arguments` releases anyone from duty of adding this each time. You just

have to put this function at the end of function `-a`, with several parameters, and subargument will be added automatically to `-b` subargument list.

Visualization of this example:

We have:

`/app/reports/` folder

`/reports` folder contain 3 text files: `general`, `john`, `thomas`.

Each of text file will be loaded from function `-b`, after choosing by user submenu option. All submenu are stored in `MenuConfig` class within dictionary `submenu_arguments_list` as:

```
'b': [ 'general', 'john', 'thomas' ];
```

`-a` function allows user to copy new file created by Frank. File name is `frank`. Function `-a` is copying file `frank` to `/reports` folder. But now frank should also add his file to submenu argument list, if he wants to lunch it in –b function. And here function will do this for him.

When you add function to the end of `-a` function, value will be added automatically to `-b` subargument list.

So when Frank add his `frank` file using `-a` option, in `MenuConfig` class within dictionary `submenu_arguments_list` will be changed record:

from: `'b': [ 'general', 'john', 'thomas' ];`

to: `'b': [ 'general', 'john', 'thomas', 'frank' ];`

Example of function usage:

```
main_file_name = 'main.py'
value = 'frank'
self.menu_object.add_value_to_submenu_arguments(main_file_name, value,
do_nod_add_repeating_values=False,  argument_name='c')
```

`Where:`

- `main_file_name –` name of main file, where you `have MenuConfig` class. If you want to change name of file of your application, also here will be given by you name. Value have to be String
- `value –` contains the value that will be added to submenu of argument. Value have to be String.
- `do_nod_add_repeating_values –` Possible values `True / False`. If set as `True`, will check if value that you trying to add already exist then will not add it again (will not duplicate values in subargument list. If set as `False` then will add value regardless if value exist in subment list or not.
- `argument_name='c' –` argument stores as Char character one letter, which is information for which argument subargument have to be add.

**WARNING.**

Please be aware that any argument for which you want use this function have to exist in `submenu_arguments_list` in `MenuConfig` class. May be existing with several values written:

```
'c': [ 'sub-argument 1', 'sub-argument 2', 'sub-argument 3' ],
```

or as empty subargument list:

```
'g': []
```

If you change the syntax of records in dictionary `submenu_arguments_list` then function probably will not work / throw an error.

# 5 Contact

If you have any questions, concerns or maybe you found bug within the code, then please do not hesitate to contact with the author

**E-mail:** tomasz.wojcik.88@gmail.com