ToDonePy

Release 4.0.9

Ryan B Patterson-Cross

Sep 17, 2020

CONTENTS:

CHAPTER

ONE

TODONEPY - A BASIC COMMAND-LINE TAST MANAGER

1.1 Introduction

Move your ToDo's to ToDone's!

ToDonePy is a command-line interface for managing your to do list. It provides a root command, *to*, and three subcommands:

- *to do* adds a new task to your list at different priorities.
- *to doing* shows you what you should be doing.
- *to done* removes a completed tast from your list.

1.2 Docs and Code

The documentation lives at https://ToDonePy.readthedocs.io/ .

The code lives at https://github.com/rbpatt2019/ToDonePy/ .

1.3 Installation

This project has been released on PyPI, so it can be installed with *pip*:

```
pip install -U ToDonePy
```

Alternatively, you can install the project manually by cloning the repo, and using the included Makefile.

```
git clone https://github.com/rbpatt2019/ToDonePy/
make install
```

If you would like to contribute to development, the install instructions are slightly different. Please see the section on contributing.

1.4 Usage

1.4.1 The base command *to*

The base command *to* has a few useful features of its own. To see what version of the command you are using, call:

```
to --version
```

As with any good command-line tool, you can get some basic help by calling:

```
to --help
```

You can get help on any subcommand by calling *–help* after that subcommand. For example, to get help with *to doing*, call:

```
to doing --help
```

Under the hood, *to* creates a Filer object that holds the information on the file you use for tracking you're TODOs. If you don't specify a file to use, it will default to *$HOME/.TODO.tsv*. If you would like to specify a different file to use, than call the command with the *–file/-f* flag like so:

```
to --file /path/to/your/TODO.tsv subcommand
```

Note: If you plan to use a file other than the default, I recommend setting it by creating the environmental variable, *TODO_LIST*.

Regardless of whether you use the default or not, calling *to* with any of the subcommands - *do*, *doing*, or *done* - will check to see if the file exists. If it does exist, *to* then pass the path on to the subcommand. If it doesn't exist, then *to* creates an empty file which it then passes on to the subcommand.

As a final note, it is worth emphasising that the contex object is only created when *to* is invoked with a subcommand. So, after a clean install, calling *to –help* or *to –version* will NOT create your *TODO.tsv* file, even if you pass the *–file/-f* flag. However, call *to do*, and it will pop into existence.

1.4.2 Adding new tasks with *to do*

To begin tracking your TODOs, call the command as follows:

```
to do rank tasks
```

to is the base command. It must be invoked to use any part of the tool. The *do* subcommand is how you add tasks to your *TODO.tsv*. After *to do*, there are two mandatory arguments: *rank* and *tasks*. The first argument is *rank*. *rank* should be a number indicating how important this task is. 1 is very important, 2 less so, etc. Though nothing explicitly bans you from using as many ranks as you want, I would reccomed using 3 for high, medium, and low priority.

The second argument is *tasks*. Here, specify what it is you need to do. If your task takes more than one word to describe, then you need to include it in quotes. *tasks* supports an indefinite number of arguments, from 1 to as many as you want.

Note: All tasks specified will be added at the same rank, so only combine tasks you want to give the same priority.

So, if you wanted to remind yourself to write an abstract for that paper you have been delaying and to email your boss, call:

```
to do 1 'Write my abstract' 'Email boss'
```

This will create *TODO.tsv* if it does not already exist, and add 'Write my abstract' and 'Email boss', both with a rank of one, to *TODO.tsv*. *to do* also logs the date and time the task was added, so that you always know how old a task is.

Sometimes, you want to sort your tasks as you add them. You can do that with the *–sort/-s* option. This specifies how to sort your list after a new task is added. It must be one of: *[rank, date, both, none]*. *both* sorts by rank and then date, and *none* does not sort, simply appending tasks to the end of your list. It defaults to *none*, on the grounds its better not to do something unless you ask. Explicit is better than implicit, as they say. If you just wanted to sort by date after adding a new task, then you could call:

```
to --sort do date 1 'Important work'
```

Note: *–sort* follows the root command *to* as it directly impacts the file and is an option accessible to all subcommands.

1.4.3 Keeping track of tasks with *to doing*

Once you have added some TODOs to your list, you need to make sure you stay on top of them. To see what needs to be done, call:

```
to doing
```

This should echo the 5 tasks at the top of your *TODO.tsv* to the terminal.

You can specify how to sort your tasks by passing the *–sort/-s* flag with one of: *[rank, date, both, none]*. It defaults to *none*, thus preserving the order in your *TODO.tsv*. Any call to sort will also change the order currently in your *TODO.tsv*, not just the order they are echoed.

Also, specifying the *–number/-n* flag will let you change how many tasks are returned, and it defaults to 5. So, if you want to return 3 tasks sorted by rank, call:

```
to -s rank doing -n 3
```

Note: Remember, *-s* is a root command option!

Maybe you prefer a graphic reminder instead of echoing in the terminal - I find this useful for spawning reminders while I am coding in VIM. *ToDonePy* has that covered, too! Just call:

```
to doing --reminder
```

to trigger a notification window. By default, it stays up for 5 seconds. Currently, you can not set the time, though that's in the works!

Note: The graphic flag makes a system call to *notify-send*. If you don't have that installed, the command will fail. It should be installed on most Linux systems, though.

Sometimes, you might want to correct an error, change a priority, or in some way edit yout *TODO.tsv*. In these cases, you can call *to doing* in editor mode:

```
to doing --edit
```

This will open *TODO.tsv* in your system editor. Where you would see something like below, if you have been following along:

```
ID      Rank    Date    Task
1       1       YYYY-MM-DD HH:MM     Write my abstract
2       1       YYYY-MM-DD HH:MM     Email boss
3       1       YYYY-MM-DD HH:MM     Important work
```

Nothing fancy, just a plain tsv with *ID* in the first column, *rank* in the second column, the date/time of addition in the third, and *task* in the fourth. Now, you can make all the changes you want, then save and close the file to return to the command line.

Calling *–edit* will trump any calls to *sort* or *number* made in the same command.

This call opens the default editor on your system, usually defined by the environmental variable EDITOR for Linux systems. If this variable is undefined, then it defaults to VIM - which should be your choice anyways! :P If that command is not found, then it will thros an OSError.

1.4.4 Completing your tasks with *to done*

After the end of a productive work session, you have completed a task from your list. Boom! Time well spent. To remove it from your *TODO.tsv*, call:

```
to done tasks
```

As with *to do*, *to done* suports an indefinite number of tasks, as long as all multi-word tasks are enclosed in quotes. For example, if you emailed your boss that finished abstract, then you can remove those tasks like so:

```
to done 'Write my abstract' 'Email boss'
```

If *to done* finds these tasks in your *TODO.tsv*, it'll remove them! If it can't find the tasks, it will print a message saying which ones couldn't be removed.

Under the hood, *to done* creates a temp file, then performs a string match to each line of your *TODO.tsv*. If a perfect match to ''task" is not in a line, that line is written to the temp file. If ''task" is in a line, that line is skipped. This

way, the temp file ends up containing only those tasks that aren't completed. Once every line is checked, the temp file replaces *TODO.tsv* with its contents. Task deleted!

> **Warning:** If two different tasks contain the same text, they will both be deleted!

1.5 Known Bugs

- Test hang when testing

1.6 Recent Changes

Please see the CHANGELOG

1.7 Next Steps

- Addition of TODOs from file parsing
- Support removal of tasks by task ID number
- Full, OS-independent graphic interface

CHAPTER

TWO

TODONEPY COMMAND-LINE INTERFACE (CLI)

2.1 Root Command: to

The root *to* command

The root command provides several options.

-s/–sort allows you to specify how to sort added or returned tasks. Bear in mind that this sorts the underlying file! It defaults to 'none' - best not to do anything unless you need to!

-f/file allows you to specify where to store your TODOs. If you don't specify, it defaults to *~/.todo.tsv* and will create the file if it doesn't exists. If you want to keep your file elsewhere, you can specify that with the env var TODO_FILE

Like all good CLIs, *-v/–version* returns the version while *-h/–help* help for the root command. Help for the subcommands can be found by calling *-h* after a subcommand, like this: *to do -h*.

`commands.to.__version__`
: The version number, pulled from the pyproject.toml file

  Type str

`commands.to.__todo__`
: The Filer object containing the TODOs. It first checks to see if the env var TODO_FILE is set. If it is, it looks there. If not, it defaults to ~/.todo.tsv. A hidden file is used to prevent clutter.

  Type *Filer*

2.2 Sub-Command: do

`subcommands.do.do`(*args: argparse.Namespace*) → None
: Add some tasks to your list

  do supports an unlimited number of tasks, but requires that tasks of more than 1 word in length be enclosed in quotes. Single or double are fine - use whichever! To keep track of how long tasks have been on the list, a timestamp of the form %y-%m-%d %H:%M is also added.

Notes

All tasks added at the same time will be added at the same rank. If you need to add multiple tasks at different ranks, you must call *to do* multiple times.

Parameters

- **args** (*argparse.Namespace*) – Arguments forwarded from the CLI. For this subcommand, this includes:
- **args.file** (*Filer*) – The TODO file to add to. From the root *to* command
- **args.rank** (*int*) – The importance to assign the new tasks.
- **args.sort** (*Literal["rank", "date", "both", "none"]*) – How to sort new tasks added to the list. From the root *to* command
- **args.tasks** (*List[str]*) – The task(s) to add to your list

Returns *None* – However, a confirmation message will be echoed to the terminal

Examples

$ to -s rank do 2 "An example task" "I'm very busy"

2.3 Sub-Command: doing

`subcommands.doing.`**`doing`**(*args: argparse.Namespace*) → None

See tasks in your list

Notes

–edit opens whatever editor is specified by your *EDITOR* env var. If one is not set, it will default to Vim.

Currently, *–reminder* has a dependency on *notify-send*. If this command is absent from your system, it will failt

Parameters

- **args** (*argparse.Namespace*) – Args passed from argparse. For this subcommand, these include:
- **args.file** (*Filer*) – The TODO file to echo. Derived from the root *to* command
- **args.sort** (*Literal['both', 'none', rank', 'date']*) – How to sort echoed tasks. Derived from the root *to* command
- **args.number** (*int*) – How many tasks to return
- **args.reminder** (*bool*) – Whether to use notify-send to create a pop-up
- **args.edit** (*bool*) – Whether to laucn an editor with your TODO file

Returns *None*

Example

$ to doing -n 3

2.4 Sub-Command: done

The *done* subcommand for the *to* main command

`subcommands.done.`**`done`**(*args: argparse.Namespace*) → None

Remove a task to your list

This command uses the supplied tasks to look for matches in your TODO list. A helpful message lets you know if the task(s) was(were) found and deleted.

Note: If your task contains more than one word, then each task must be enclosed in quotes. Otherwise, the CLI treats each word as a task. Also note that if multiple lines match a task, they will ALL be deleted.

Parameters

- **args** (*argparse.Namespace*) – Arguments forwarded from the CLI. For this subcommand, this includes:
- **args.file** (*Filer*) – The TODO file to be searched. From the root *to* command
- **args.task** (*List[str]*) – The list of tasks to be deleted

Returns *None* – Though a message will be echoed letting you know if the task(s) was(were) deleted successfully.

Example

$ to done 'An example' 'Is always helpful'

CHAPTER

THREE

TODONEPY HELPERS

3.1 ToDonePy Helper Functions

3.1.1 Function: itemsetter

helpers.itemsetter.**itemsetter**(**items: int*) → Callable[[List, Any], None]

Return a callable object that sets item from its operand

This is essentially the opposite of operator.itemgetter. If only one position is specified, the resulting callable will set that item. If multiple positions are specified, it sets all items

Parameters ***items** (*int*) – The indices to be set. Remember, Python is 0-indexed

Returns *Callable[[List, Any], None]* – A function that will set the indices specified in *items* to a given value.

Examples

```
>>> x = ['a', 'b', 'c']
>>> f = itemsetter(2)
>>> f(x, 'z')
>>> print(x)
['a', 'b', 'z']
```

3.1.2 Function: external_command

helpers.external_command.**external_command**(*args: List[str]*) → subprocess.CompletedProcess

Make a generic command line call

Any command line call can be made. Pass the respective components as individual strings. Roughly speaking, anywhere there is a space, break it into a new component. See the documentation on subprocess.run for advanced use cases.

Note: If run in a situation where the user was providing a dynamic input, there are obvious security risks. In the app, however, the user cannot provide their own input, which I believe sufficiently mitigates the risk in this use case. Obviously, if you adopt and use this function elsewhere, take care to check your inputs!

Parameters ***args** (*List[str]*) – The parts of the external command

Returns *subprocess.CompletedProcess* – If successful. This contains a number of useful attributes, including returncode and stdout.

Raises

- **OSError** – If unsuccessful. This will be thrown if the command found in args[0] cannot be found on the OS
- **subprocess.CalledProcessError** – If the called command returns a non-zero exit status

Examples

The results of a successful command ccan be queried like so:

```
>>> results = external_command(['echo', 'hello'])
>>> results.returncode
0
```

3.2 The *Filer* Class

class helpers.filer.**Filer**(*path: pathlib.Path*, *create: bool = True*, *delimiter: str = '\t'*)

Bases: object

A class for gracefully handling file interactions with delimited data

Designed particularly for passing context in a CLI, it is a thin wrapper for many common file I/O actions, including reading, writing (both lines and columns), and deleting.

append(*rows: List[List[str]]*) → None

Appends contents of *rows* to self.path

Note: This will not over-write the contents of the file, mirroring the modes of open()

Parameters **rows** (*List[List[str]]*) – A list of strings to write to self.path.

Returns *None*

Examples

```
>>> example.append([['f','g', 'h'], ['i', 'j', 'k']])
```

delete(*contains: str*) → bool

Deletes all lines from self where *contains in line*

Parameters **contains** (*str*) – String to match for line deletion

Returns *bool* – True if successulf, false otherwise

Example

```
>>> example.delete('j')
True
```

read() → List[List[str]]
Read the lines of self.path

Note: Reads in all lines, so will suffer on large files

Parameters None

Returns *List[List[str]]* – A list of lines where each line is a list of column values

Examples

```
>>> example.read()
[['ID', 'Rank', 'Date', 'Task'], ['f', 'g', 'h']]
```

sort(*cols: List[int]*, *header: bool = False*) → None
Sort the contents of self.path by columns

Parameters

- **cols** (*List[int]*) – List of column indices indicating what to sort by. Remember, Python is 0-indexed
- **header** (*bool*) – Whether or not row 0 is a header. If True, row 0 is skipped for sorting

Returns *None*

Example

```
>>> example.sort([1, 2], header=False)
```

write(*rows: List[List[str]]*) → None
Writes contents of rows to self.path.

Warning: If the file already has content, that will be overwritten! This mirrors the modes used by open()

Parameters **rows** (*List[List[str]]*) – A list of strings to write to self.path. *rows[0]* represents line 1, and *rows[0][0]* is line 1, column 1.

Returns *None*

Examples

```
>>> example.write([['a', 'b', 'c']])
```

write_col(*col: List[str]*, *index: int = 0*) → None
Writes contents of col to self.path at specified index

> **Warning:** If the column already has content, that will be overwritten! This mirrors the modes used by open()

Parameters

- **col** (*List[str]*) – A list of strings to write to self.path. This should be the same length as *self.length*
- **index** (*int*) – Which column to write at. Remember, Python is 0-indexed.

Returns *None*

Raises **IndexError** – When *col* has more or less items than *self.length*

Examples

```
>>> example.write_col(['d'], index=2)
```

CHAPTER

FOUR

TESTING

4.1 Configurations

conftest.**doctest_filer_example**(*doctest_namespace: Dict[str, helpers.filer.Filer]*, *tmp_path: pathlib.Path*) → None

Fixture for instantiating an example Filer for use in doctests

Parameters

- **doctest_namespace** (*Dict[str, Filer]*) – *pytest.fixture* holding variables to be used in doctests
- **tmp_path** (*Path*) – *pytest.fixture* containing a temporary file path

Returns *None*

conftest.**tmp_file**(*tmp_path: pathlib.Path*) → *helpers.filer.Filer*

Fixture for automating setup of files

Parameters **tmp_path** (*Path*) – pytest.fixture. Where to create the file

Returns *Path* – An instantiated tsv file

4.2 Test Modules

4.2.1 Testing the do sub-command

test_do.**test_to_do**(*sort: str*, *expected: str*, *tmp_file:* helpers.filer.Filer, *capsys*)

Check that tasks are appropriately added and sorted

Parametrized to check various calls to the *–sort* flag

4.2.2 Testing the doing sub-command

test_doing.**test_to_doing**(*sort: str*, *expected: str*, *tmp_file:* helpers.filer.Filer, *capsys*)

Run to doing with existing custom file

Parametrised to test situations where *–sort* is/isn't passed

test_doing.**test_to_doing_custom_file_edit_flag**(*tmp_file:* helpers.filer.Filer, *capsys*)

Run to doing with the edit flag

test_doing.**test_to_doing_custom_file_graphic_flag**(*tmp_file*, *capsys*)

Run to doing with the *–reminder* flag

4.2.3 Testing the done sub-command

test_done.**test_to_done**(*tmp_file:* helpers.filer.Filer, *capsys*)
Check that task are appropriately deleted from the TODO file

4.2.4 Testing the Filer class

test_filer.**test_Filer_append_existing_file**(*tmp_file:* helpers.filer.Filer) → None
Run Filer to append to an existing file

test_filer.**test_Filer_create**(*tmp_path:* *pathlib.Path*, *create:* *bool*, *expected:* *List[List[Union[None, str]]]*) → None
Run Filer to read a file that does not exist

This is parametrize to test that if fails if *create* = *False* but passes when *create* = *True*

test_filer.**test_Filer_delete_existing_file**(*tmp_file:* helpers.filer.Filer, *to_del:* *str*, *expected: List[List[str]]*) → None
Run Filer to delete a line from an existing file

This is parametrised to check conditions where a line is not deleted

test_filer.**test_Filer_read_existing_file**(*tmp_file:* helpers.filer.Filer) → None
Run Filer to read an existing file

test_filer.**test_Filer_sort_existing_file**(*tmp_file:* helpers.filer.Filer, *header:* *bool*, *expected: List[List[str]]*) → None
Run Filer to sort an existing file

This is parametrised to check that headers are treated properly

test_filer.**test_Filer_write_col_error**(*tmp_file:* helpers.filer.Filer) → None
Check that *Filer.write_col* raises an *IndexError* if col is the wrong length

test_filer.**test_Filer_write_existing_file**(*tmp_file:* helpers.filer.Filer) → None
Run Filer to write to an existing file

4.2.5 Testing Helper Functions

test_external_command.**test_ec_OSError**() → None
Test *external_command* raise an *OSError* for *Command Not Found*

test_external_command.**test_ec_ProcessError**() → None
Test *external_command* raise an CalledProcessError when has a non-0 status

test_external_command.**test_ec_successful**() → None
Test a basic command call with *external_command*

Checks that a successful call returns an exit code of 0 and the expected output

test_itemsetter.**test_itemsetter**() → None
Test itemsetter with basic inputs

Returns None

CHAPTER

FIVE

FOR CONTRIBUTORS

Comments, criticisms, and concerns are always welcome! If you would like to help with development, please follow the steps below. This project depends on Poetry for all things dependency and development related. Make sure it's installed, or else all this will fail. It's an awesome tool, I highly recommend you check it out!

5.1 Clone the repo

```
git clone https://github.com/rbpatt2019/ToDonePy.git
cd ToDonePy
```

5.2 Make a new environment

Follow your own protocol! I use pyenv for all my env/venv control, so I would do:

```
pyenv virtualenv ToDonePy
pyenv local ToDonePy
```

Regardless of how you do it, run the following once its created:

```
make develop
```

5.3 Start developing

Checkout the Makefile for lots of useful commands for testing, linting, and many others! Before committing any changes, I'd strongly recommend creating a new branch:

```
git checkout -b new_feature
```

5.4 And contribute!

Once you're ready to share your changes, fork the repository on github. Then, add it as a remote to the repo and push the changes there.

```
git remote add origin https://github.com/YOUR_USER/ToDonePy.git
git push origin new_feature
```

Finally, open a pull request, and I'll review it as soon as I can!

If you're a command line nut like me, this can all be done from the command line using hub, a CLI for interacting with the github api. See their repo for installation instructions. Instead of the above, do:

```
hub fork --remote-name=origin
git push origin new_feature
hub pull-request
```

This will fork the repo, push your changes, and create a pull request, all without leaving the command line!

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

F

H

I

M

R

S

T

W