PARSEARG: TURNS ARGPARSE ON ITS HEAD, THE DECLARATIVE WAY

THOMAS P. HARTE

Contents

1. Overview

The "standard" Python module for writing command-line interfaces ("CLI") is `argparse`. It is standard in so far as it is one of the batteries included with the Python distribution, so no special installation is required. There are many other packages for creating CLIs, probably because `argparse` is a bit clunky to use. Putting together a CLI with `argparse` alone is nothing if not an exercise in imperative programming, and this has three very negative consequences:

(1) It obfuscates the intention of the CLI design;

(2) It is prone to errors;

(3) It discourages CLI design in the first instance; it makes debugging a CLI design very difficult; and it makes refactoring or re-configuring the CLI design overly burdensome.

The `parsearg` package is nothing more than a layer over `argparse` that exposes the `argparse` functionality via a `dict`. The `dict` is the view component of the model-view-controller ("MVC") design pattern. The `dict` embeds callbacks to the Controller component, thereby achieving a clean separation of duties, which is what the MVC pattern calls for. By separating the View component into a `dict`, the CLI design can be expressed in a declarative way: `parsearg` manifests the *intention* of the CLI design without having to specify how that design is implemented in terms of `argparse`'s parsers and subparsers.

Other packages—such as `click` and `plac`—effectively decorate functions that are part of the Controller with functionality from the View. Unfortunately, while this may expose the functionality of `argparse` in a more friendly way via the decorators, it dissipates the elements of the View across the Controller and makes the CLI design difficult to grasp.

The `parsearg` philosophy is that `argparse` is already good enough in terms of the functionality that it provides, but that it just needs a little nudge in terms of how it's used. Arguments to be added to a CLI with `argparse` can be clearly—and solely—specified as data, as can the callbacks that consume these arguments. `parsearg` takes advantage of this by specifiying everything (in the View component of MVC) as a `dict`, from which `parsearg` then generates a parser (or set of nested parsers) using `argparse` which the Controller then uses. The declarative nature of the `parsearg` approach places the CLI design front and center via a `dict` (one of Python's built-in data structures), the keys of which specify a flattened tree that `parsearg` renders as an unflattened tree of `argparse` parsers. `parsearg` requires nothing special: It works with Python out of the box, and therefore uses what's already available without introducing dependencies.

Simple? The following examples should help.

Date: 2020-11-23 Build: `v-0.1.5 (3b78c08635c23f360583a709fe490aed90609978)`.

2. `todos`

Yet another To-Do app? Yes. It illustrates:

(1) A sufficiently realistic, but not overly complex, problem;

(2) The MVC pattern in the wild;

(3) How `parsearg` neatly segments the View component into a `dict`.

Let's start from the outside of the onion. How do we interact with `todos.py`?

```
#python todos purge users
#python todos purge todos

python todos.py create user foo -e foo@foo.com -p 212-555-1234
python todos.py create user bar -e bar@bar.com
python todos.py create user qux -p 212-123-5555

python todos.py create todo foo title1 -c description1 -d 2020-11-30
python todos.py create todo foo title2 -c description2 --due-date=2020-12-31
python todos.py create todo qux todo-1 --description=Christmas-party -d 2020-11-30
python todos.py create todo qux todo-2 --description=New-Year-party

python todos.py update user email qux qux@quxbar.com
python todos.py update user phone bar 203-555-1212

python todos.py update todo title 4 most-important
python todos.py update todo description 4 2021-party

python todos.py show users
python todos.py show todos
```

```
(base) tharte@trex:~/dot/py/python/parsearg/parsearg/examples$ (base) tharte@trex:~/dot/py/python/parsearg/parsearg/examples$ 'create user fo
------------------------------------------------
        None
'create user bar -e bar@bar.com':
---------------------------------
        None
'create user qux -p 212-123-5555':
----------------------------------
        None
(base) tharte@trex:~/dot/py/python/parsearg/parsearg/examples$ 'create todo foo title1 -c description1 -d 2020-11-30':
------------------------------------------------------------
        None
'create todo foo title2 -c description2 --due-date=2020-12-31':
---------------------------------------------------------------
        None
'create todo qux todo-1 --description=Christmas-party -d 2020-11-30':
---------------------------------------------------------------------
        None
'create todo qux todo-2 --description=New-Year-party':
------------------------------------------------------
        None
(base) tharte@trex:~/dot/py/python/parsearg/parsearg/examples$ 'update user email qux qux@quxbar.com':
---------------------------------------
        None
'update user phone bar 203-555-1212':
-------------------------------------
        None
(base) tharte@trex:~/dot/py/python/parsearg/parsearg/examples$ 'update todo title 4 most-important':
-------------------------------------
        None
'update todo description 4 2021-party':
---------------------------------------
        None
(base) tharte@trex:~/dot/py/python/parsearg/parsearg/examples$ 'show users':
-------------
        None
'show todos':
-------------
        None
```

This populates two tables—`User` and `Todo`—in a SQLite database:

```
sqlite3 todo.db 'select * from User;'
```

```
foo|foo@foo.com|212-555-1234|2020-11-23 11:48:38
bar|bar@bar.com|203-555-1212|2020-11-23 11:48:38
qux|qux@quxbar.com|212-123-5555|2020-11-23 11:48:38
```

```
sqlite3 todo.db 'select * from Todo;'
```

```
1|title1|description1|2020-11-30|2020-11-23 11:48:39|foo
2|title2|description2|2020-12-31|2020-11-23 11:48:39|foo
3|todo-1|Christmas-party|2020-11-30|2020-11-23 11:48:39|qux
4|most-important|2021-party|None|2020-11-23 11:48:39|qux
```

Let's look at the Python code for `todos.py`.

We can generate a tree view of the CLI design as follows:

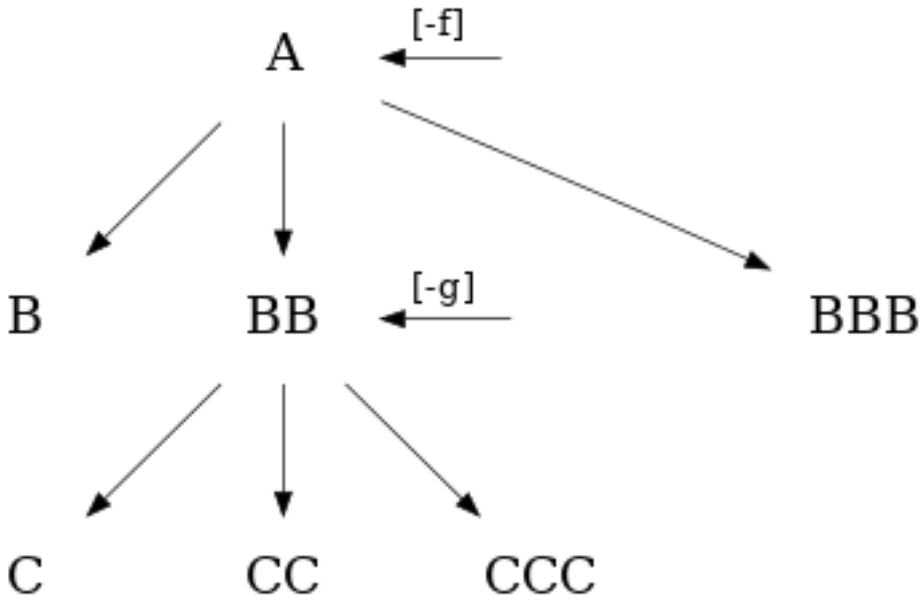
```
from parsearg.examples.todos import view
from parsearg.parser import ParseArg

print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
TODO
    update
        user
            phone
            email
        todo
            title
            description
    show
        todos
        users
    purge
        todos
        users
    create
        user
        todo
```

3. How PARSEARG works: Illustration with the A-tree and the A-AA tree

The "A tree" example (simpler form of the "ABC" example). Note that `[-f]` and `[-g]` here indicate optional arguments that attach to the nodes (`A` and `BB`, respectively): Why "optional"? Because `B`, `BB` and `BBB` are *positional arguments*: once a positional argument is in place it must be attributed to one of `B`, `BB` or `BBB`, so anything associated with `A` must be optional by definition. This also means that `[-f]` and `[-g]` should *not* be interpreted as child nodes of `A` and `BB`, respectively.



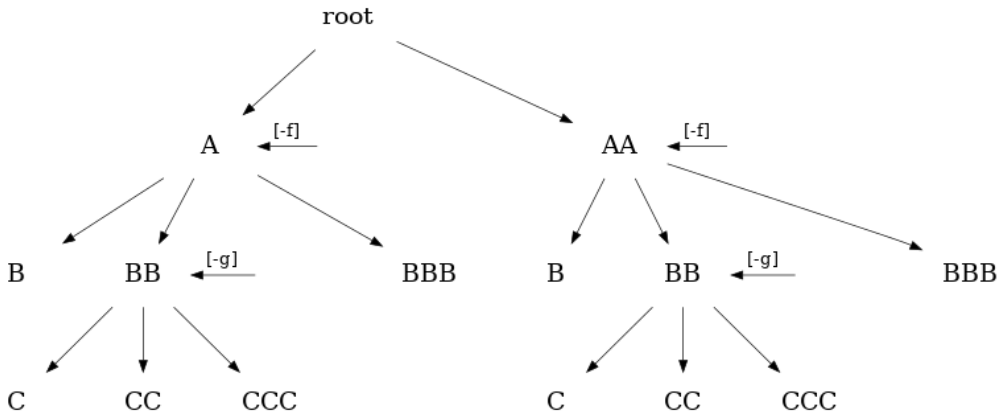
Let's look at the Python code for `a.py`.

```
from parsearg.examples.a import view
from parsearg.parser import ParseArg

print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
TODO
    A
        B
        BB
            C
            CC
            CCC
        BBB
```

The extended "ABC" example (this is the "A-AA" tree):



Let's look at the Python code for a_aa.py.

```python
from parsearg.examples.a_aa import view
from parsearg.parser import ParseArg

print(
    ParseArg(d=view, root_name='TODO').tree.show(quiet=True)
)
```

```
TODO
    A
        B
        BB
            C
            CC
            CCC
        BBB
    AA
        B
        BB
            C
            CC
            CCC
        BBB
```

Let's examine parsearg's ~parser.py.Python code for parser.py.