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 that comes included with the Python distribution, so no special installation is required. Probably because `argparse` is a bit clunky to use. many other (non-standard) packages have been developed for creating CLIs. Why "clunky"? 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.

In spite of this clunkiness, `argparse` has everything we need in terms of functionality. `parsearg`, then, 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 within 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 (`parsearg` does that for you).

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 packages' decorators, it dissipates the elements of the View across the Controller and in so doing it 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 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 this `dict` form a flattened tree of sub-commands. `parsearg` unflattens the flattened tree into a 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.

Date: 2020-11-23 Build: `v-0.1.9 (397408f71022223facf3272e1a1e71f3a049219b)`.

Simple? The following examples should help.

2. TODOS

Yet another To-Do app? Yes, because:

(1) It illustrates a sufficiently realistic, but not overly complex, problem;

(2) It also illustrates the MVC pattern in the wild;

(3) It shows how `parsearg` neatly segments the View component into a `dict`.

Let's start with the outer layer of the onion. How do we interact with `todos.py`? First, create some users in the `User` table with the `create user` subcommand of `todos.py`. Note that we do not (yet) have a phone number for user `bar`, nor do we have an email address for user `qux`:

```
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
```

```
Traceback (most recent call last):
main(' '.join(args))
  File "todos.py", line 122, in main
    result = ns.callback(ns)
  File "todos.py", line 25, in create_user
    model.User().create(
  File "/home/tharte/dot/py/python/parsearg/parsearg/examples/model.py", line 164, in create
    result = self._conn.execute(sql)
sqlite3.IntegrityError: UNIQUE constraint failed: User.name
Traceback (most recent call last):
main(' '.join(args))
  File "todos.py", line 122, in main
    result = ns.callback(ns)
  File "todos.py", line 25, in create_user
    model.User().create(
  File "/home/tharte/dot/py/python/parsearg/parsearg/examples/model.py", line 164, in create
    result = self._conn.execute(sql)
sqlite3.IntegrityError: UNIQUE constraint failed: User.name
Traceback (most recent call last):
main(' '.join(args))
  File "todos.py", line 122, in main
    result = ns.callback(ns)
  File "todos.py", line 25, in create_user
    model.User().create(
  File "/home/tharte/dot/py/python/parsearg/parsearg/examples/model.py", line 164, in create
    result = self._conn.execute(sql)
sqlite3.IntegrityError: UNIQUE constraint failed: User.name
```

Second, create some to-dos in the `Todo` table with the `create todo` subcommand of `todos.py`. Note that

```
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
```

```
'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
```

Let's make some changes to the records entered so far. We can add an email address for user `qux` and a phone number for user `bar`:

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

```
'update user email qux qux@quxbar.com':
----------------------------------------
        None
'update user phone bar 203-555-1212':
--------------------------------------
        None
```

Now update two of the to-dos, changing the `title` and the `description` in the fourth to-do:

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

```
'update todo title 4 most-important':
-------------------------------------
        None
'update todo description 4 2021-party':
----------------------------------------
        None
```

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

```
'show users':
-------------
        None
'show todos':
-------------
        None
```

The result of these commands is that the two tables (`User` and `Todo`) are populated in a SQLite database:

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

```
foo|foo@foo.com|212-555-1234|2020-11-23 22:02:58
bar|bar@bar.com|203-555-1212|2020-11-23 22:02:58
qux|qux@quxbar.com|212-123-5555|2020-11-23 22:02:59
```

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

```
1|title1|description1|2020-11-30|2020-11-23 22:03:02|foo
2|title2|description2|2020-12-31|2020-11-23 22:03:02|foo
3|todo-1|Christmas-party|2020-11-30|2020-11-23 22:03:02|qux
4|most-important|2021-party|None|2020-11-23 22:03:02|qux
5|title1|description1|2020-11-30|2020-11-23 22:24:15|foo
6|title2|description2|2020-12-31|2020-11-23 22:24:15|foo
7|todo-1|Christmas-party|2020-11-30|2020-11-23 22:24:15|qux
8|todo-2|New-Year-party|None|2020-11-23 22:24:16|qux
9|title1|description1|2020-11-30|2020-11-23 22:42:06|foo
10|title2|description2|2020-12-31|2020-11-23 22:42:06|foo
11|todo-1|Christmas-party|2020-11-30|2020-11-23 22:42:06|qux
12|todo-2|New-Year-party|None|2020-11-23 22:42:06|qux
```

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

The View component is entirely contained within a single `dict`, *viz.* `view`, which has been formatted here for clarity using `parsearg.utils.show`:

```
from parsearg.examples.todos import view
from parsearg.utils import show

show(view)
```

```
'purge|users':
    'callback':
    <function purge_users at 0x7f535ca1dca0>
'purge|todos':
    'callback':
    <function purge_todos at 0x7f535ca298b0>
'show|users':
    'callback':
    <function show_users at 0x7f535ca29940>
'show|todos':
    'callback':
    <function show_todos at 0x7f535ca299d0>
'create|user':
    'callback':
    <function create_user at 0x7f535ca29a60>
    'name':
    {'help': 'create user name', 'action': 'store'}
    '-e|--email':
    {'help': "create user's email address", 'action': 'store', 'default': ''}
    '-p|--phone':
    {'help': "create user's phone number", 'action': 'store', 'default': ''}
'create|todo':
    'callback':
    <function create_todo at 0x7f535ca29af0>
    'user':
    {'help': 'user name', 'action': 'store'}
    'title':
    {'help': 'title of to-do', 'action': 'store'}
    '-c|--description':
    {'help': 'description of to-do', 'action': 'store', 'default': ''}
    '-d|--due-date':
    {'help': 'due date for the to-do', 'action': 'store', 'default': None}
'update|user|email':
    'callback':
    <function update_user_email at 0x7f535ca29b80>
    'name':
    {'help': 'user name', 'action': 'store'}
    'email':
    {'help': 'user email', 'action': 'store'}
'update|user|phone':
    'callback':
    <function update_user_phone at 0x7f535ca29c10>
    'name':
    {'help': 'user name', 'action': 'store'}
    'phone':
    {'help': 'user phone', 'action': 'store'}
'update|todo|title':
    'callback':
    <function update_todo_title at 0x7f535ca29ca0>
    'id':
    {'help': 'ID of to-do', 'action': 'store'}
    'title':
    {'help': 'title of to-do', 'action': 'store'}
'update|todo|description':
    'callback':
    <function update_todo_description at 0x7f535ca29d30>
    'id':
    {'help': 'ID of to-do', 'action': 'store'}
    'description':
    {'help': 'description of to-do', 'action': 'store'}
```

We can generate a tree view of the CLI design specified by the above `dict`, namely `view`, as follows:

```
from parsearg.parser import ParseArg

print(
```

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

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

3. How PARSEARG works

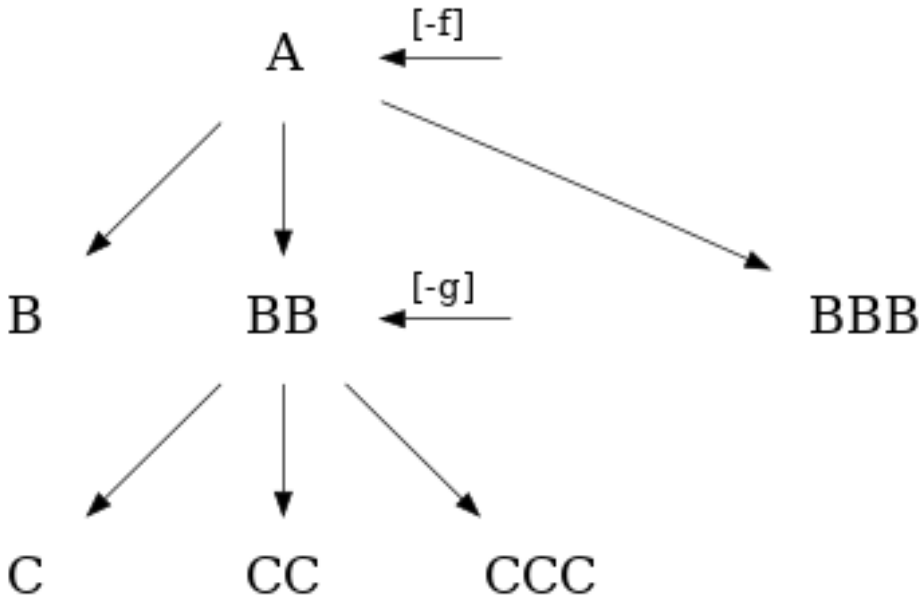
It is easier to explain how `parsearg` works with a simpler abstraction than the above example `parsearg.examples.todos.py`. Here, we will consider nested parsers as trees. We introduce two trees for this purpose:

(1) The "A tree", and

(2) The "A-AA tree".

3.1. **The A-tree.**

The "A tree" has three levels. As each node of the tree must necessarily occupy a positional argument of the command line, `[-f]` and `[-g]` must correspondingly be *optional arguments* that attach to the nodes (`A` and `BB`, respectively, in the below diagram).



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

Consider the `dict` that represents the View component:

```
from parsearg.examples.a import view

show(view)
```

```
'A':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c992550>
    '-c':
    {'help': 'A [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A verbosity', 'action': 'store_true'}
'A|B':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5bdaf0>
    '-c':
    {'help': 'A B [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A B verbosity', 'action': 'store_true'}
'A|BB':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5bdb80>
    '-c':
    {'help': 'A BB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB erbosity', 'action': 'store_true'}
'A|BB|C':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5bdc10>
    '-c':
    {'help': 'A BB C [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB C verbosity', 'action': 'store_true'}
'A|BB|CC':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5bdca0>
    '-c':
    {'help': 'A BB CC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CC verbosity', 'action': 'store_true'}
'A|BB|CCC':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5bdd30>
    '-c':
    {'help': 'A BB CCC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CCC verbosity', 'action': 'store_true'}
'A|BBB':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5bddc0>
    '-c':
    {'help': 'A BBB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BBB verbosity', 'action': 'store_true'}
```

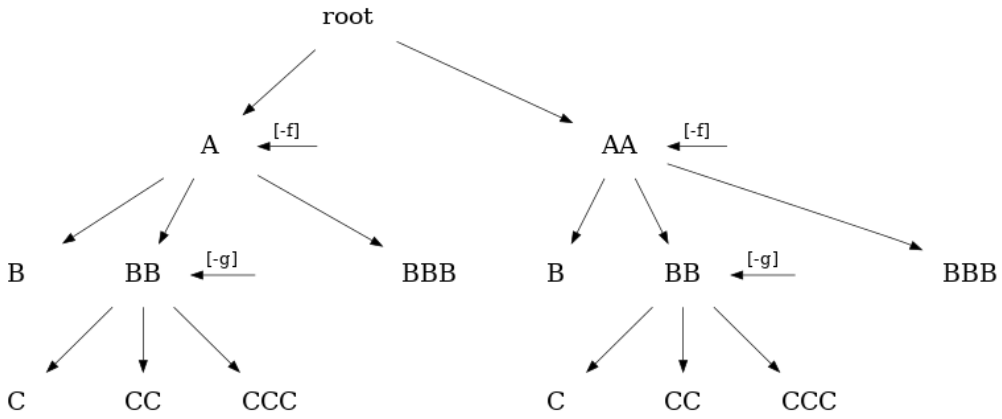
and then the tree that the parsed `dict` is represented by:

```
from parsearg.parser import ParseArg

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

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

3.2. **The A-A tree.** The A-AA tree simply extends the A tree one level:



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

Consider the `dict` that represents the View component:

```
from parsearg.examples.a_aa import view

show(view)
```

```
'A':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5d0820>
    '-c':
    {'help': 'A [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A verbosity', 'action': 'store_true'}
'A|B':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5d08b0>
    '-c':
    {'help': 'A B [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A B verbosity', 'action': 'store_true'}
'A|BB':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5d0940>
    '-c':
    {'help': 'A BB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB erbosity', 'action': 'store_true'}
'A|BB|C':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5d09d0>
    '-c':
    {'help': 'A BB C [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB C verbosity', 'action': 'store_true'}
'A|BB|CC':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5d0a60>
    '-c':
    {'help': 'A BB CC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CC verbosity', 'action': 'store_true'}
'A|BB|CCC':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c05e0>
```

```
    '-c':
    {'help': 'A BB CCC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BB CCC verbosity', 'action': 'store_true'}
'A|BBB':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c0f70>
    '-c':
    {'help': 'A BBB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'A BBB verbosity', 'action': 'store_true'}
'AA':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c0ee0>
    '-c':
    {'help': 'AA [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA verbosity', 'action': 'store_true'}
'AA|B':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c0e50>
    '-c':
    {'help': 'AA B [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA B verbosity', 'action': 'store_true'}
'AA|BB':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c0d30>
    '-c':
    {'help': 'AA BB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB verbosity', 'action': 'store_true'}
'AA|BB|C':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c0ca0>
    '-c':
    {'help': 'AA BB C [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB C verbosity', 'action': 'store_true'}
'AA|BB|CC':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c0c10>
    '-c':
    {'help': 'AA BB CC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB CC verbosity', 'action': 'store_true'}
'AA|BB|CCC':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c00d0>
    '-c':
    {'help': 'AA BB CCC [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BB CCC verbosity', 'action': 'store_true'}
'AA|BBB':
    'callback':
    <function make_callback.<locals>.func at 0x7f535c5c04c0>
    '-c':
    {'help': 'AA BBB [optional pi]', 'action': 'store_const', 'const': 3.141593}
    '-v|--verbose':
    {'help': 'AA BBB verbosity', 'action': 'store_true'}
```

and then the tree that the parsed `dict` is represented by:

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

```
TODO
    A
        BBB
        B
        BB
            CC
```

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