Argpext 1.2.2 — Documentation

Argpext is a module dedicated to improving the command line interface with Python module internals. It allows one to quickly expose any selected Python functions to the command line within DOS or Linux-like shells. Help messages are automatically produced.

Argpext provides hierarchical extension to the “Sub-commands” utility of the standard **argparse** module. It allows one to group any Python functions into a hierarchical tree-like structure, e.g. according to their logic. Every such function then corresponds to a certain sequence of sub-commands, and can be executed directly from the command line by simply passing the sequence as command line arguments to the top level script. The rest of the command line arguments to the script are used to set up the values of function arguments, at which level the standard **argparse** interface applies.

Argpext provides a special variable type to support command line arguments that take predetermined set of values. Information about available choices is automatically propagated into the usage help message.

The best way to learn Argpext is through an example. We introduce Argpext through a series of illustrative examples of tested programs. The formal *reference* is at the bottom.

Argpext is an extension of **argparse**; its knowledge is assumed in our document.

Building the command line hierarchy

In this section we build the sub-command hierarchy for an application in order to establish the efficient connection between the command line arguments and corresponding Python functions.

Bare bones example

Let us further consider as an example, how the following simplistic game involving sheep and wolves may be designed.

Suppose there is a function called **sheep_graze()** that lets the sheep graze. Here is how we can use the standard **argparse** module in order to connect this function to the command line:

```
import argparse

def sheep_graze(feed):
    print('Sheep grazes on %s' % feed)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Let sheep graze')
    parser.add_argument('-f', dest='feed', default='grass',
                        help='Specify the feed. Default: %(default)s.')
    argv = parser.parse_args()
```

```
    sheep_graze(feed=argv.feed)
```

The identical functionality is now achieved with our Argpext as follows:

```
import argpext

def sheep_graze(feed):
    print('Sheep grazes on %s' % feed)

class SheepGraze(argpext.Task):
    "Let sheep graze"
    hook = argpext.make_hook(sheep_graze)
    def populate(self,parser):
        parser.add_argument('-f', dest='feed', default='grass',
                            help='Specify the feed. Default: %(default)s.')

if __name__ == '__main__':
    SheepGraze().digest()
```

Class **SheepGraze**, constructed by inheritance from **argpext.Task**, establishes the interface between command line and function **sheep_graze()**.

Command line is processed during the call to the **SheepGraze.digest()** function.

The docstring "Let sheep graze" shows up inside the usage. Indeed, when the above program is saved as file sheepgraze.py and executed with the --help or -h switches, we have:

```
$ sheepgraze.py -h
usage: sheepgraze.py [-h] [-f FEED]

Let sheep graze

optional arguments:
  -h, --help  show this help message and exit
  -f FEED     Specify the feed. Default: grass.
```

Examples of execution

Task **sheep_graze()** can be executed from the command line as follows:

```
$ sheepgraze.py
Sheep grazes on grass

$ sheepgraze.py -f daisies
Sheep grazes on daisies
```

Equivalently, in Python interpreter:

```
>>> import sheepgraze
>>> sheepgraze.SheepGraze()()
Sheep grazes on grass

>>> sheepgraze.SheepGraze()(feed='daisies')
```

```
Sheep grazes on daisies
```

Adding a new sub-command

Suppose we now wish to add another function **`sheep_jump()`** to the *example* above.

First we should add a new class **`SheepJump`** which is completely analogous to the previously described **`SheepGraze`**.

Let us then introduce sub-commands `graze` and `jump`. In order to differentiate between the two different tasks on the level of command line.

To provide the mapping between sub-commands `graze` and `jump` and their respective implementations **`SheepGraze()`** and **`SheepJump()`** we declare class **`Sheep`** (subclass of **`argpext.Node`**) and assign the mapping to its **`SUBS`** attribute, as shown in the example below. Tasks **`SheepGraze()`** and **`SheepJump()`** are now attached to node **`Sheep`**.

The next key thing is to include **`Sheep.digest()`** at the bottom in order to execute command line on our new interface.

```
import argpext

def sheep_graze(feed):
    print('Sheep grazes on %s' % feed)

class SheepGraze(argpext.Task):
    "Let sheep graze"
    hook = argpext.make_hook(sheep_graze)
    def populate(self,parser):
        parser.add_argument('-f', dest='feed', default='grass',
                            help='Specify the feed. Default: %(default)s.')


def sheep_jump(n):
    print('Sheep jumps %d times' % n)

class SheepJump(argpext.Task):
    "Let sheep jump"
    hook = argpext.make_hook(sheep_jump)
    def populate(self,parser):
        parser.add_argument('-n', dest='n', default=2, type=int,
                            help='Specify the number of jumps')


class Sheep(argpext.Node):
    "Sheep-related tasks"
    SUBS = [('graze', SheepGraze),  # Link subcommand 'graze' to class SheepGraze
            ('jump', SheepJump),    # Link subcommand 'jump'  to class SheepJump
            # Add more subcommands here
            ]


if __name__ == '__main__':
    Sheep().digest()
```

When the above program is saved as file `sheepactions.py` and executed, we have:

```
$ sheepactions.py -h
usage: sheepactions.py [-h] {graze,jump} ...

Sheep-related tasks

positional arguments:
  {graze,jump}  Description
    graze       Let sheep graze
    jump        Let sheep jump

optional arguments:
  -h, --help    show this help message and exit
```

The sub-commands `graze` and `jump` are clearly shown in the help message. In order to display their individual usage one should pass any of these sub-commands followed by the `--help/-h` switch. For example, to display the usage for `graze`:

```
$ sheepactions.py graze -h
usage: sheepactions.py graze [-h] [-f FEED]

Let sheep graze

optional arguments:
  -h, --help  show this help message and exit
  -f FEED     Specify the feed. Default: grass.
```

Examples of execution:

In command line:

```
$ sheepactions.py graze -f daisies
Sheep grazes on daisies

$ sheepactions.py jump -n 5
Sheep jumps 5 times
```

Equivalently, in Python interpreter:

```
>>> import sheepactions
>>> from sheepactions import *
>>> SheepGraze()(feed='daisies')
Sheep grazes on daisies

>>> SheepJump()(n=5)
Sheep jumps 5 times
```

Attaching one node to another

In addition to attaching functions to a node, it is also possible to attach nodes to another

node, as demonstrated by lines 18 and 19 of the following example

```
import argpext

import sheepactions # Module sheepactions is provided by previous example.



class FeedWolf(argpext.Task):
    "Feed the wolf"

    def hook(self,prey):
        print('Wolf eats %s' % prey)

    def populate(self,parser):
        parser.add_argument('-p', dest='prey', default='sheep',
                            help='Specify the food. Default:"%(default)s".')

class Main(argpext.Node):
    "Top level sheepgame options"
    SUBS = [
        ('sheep', sheepactions.Sheep), # Attaching another Node
        ('feed-wolf', FeedWolf), # Attaching a Task
        # Add more subcommands here
        ]

if __name__ == '__main__':
    Main().digest()
```

This methodology allows one to build a rather general hierarchical tree-like structure of subcommands of non-uniform height.

When the above program is saved as file `sheepgame.py`, the top level help message is invoked as follows:

```
$ sheepgame.py -h
usage: sheepgame.py [-h] {sheep,feed-wolf} ...

Top level sheepgame options

positional arguments:
  {sheep,feed-wolf}  Description
    sheep            Sheep-related tasks
    feed-wolf        Feed the wolf

optional arguments:
  -h, --help         show this help message and exit
```

To display sheep-related usage of `sheepgame.py`, pass the `sheep` subcommand:

```
$ sheepgame.py sheep -h
usage: sheepgame.py sheep [-h] {graze,jump} ...

Sheep-related tasks

positional arguments:
```

```
  {graze,jump}  Description
    graze       Let sheep graze
    jump        Let sheep jump

optional arguments:
  -h, --help    show this help message and exit
```

To display even lower level help messages, additional sub-commands are passed:

```
$ sheepgame.py sheep jump -h
usage: sheepgame.py sheep jump [-h] [-n N]

Let sheep jump

optional arguments:
  -h, --help  show this help message and exit
  -n N        Specify the number of jumps

$ sheepgame.py sheep graze -h
usage: sheepgame.py sheep graze [-h] [-f FEED]

Let sheep graze

optional arguments:
  -h, --help  show this help message and exit
  -f FEED     Specify the feed. Default: grass.
```

Examples of execution:

In the command line:

```
$ sheepgame.py sheep jump -n 5
Sheep jumps 5 times

$ sheepgame.py sheep graze
Sheep grazes on grass

$ sheepgame.py sheep graze -f daisies
Sheep grazes on daisies
```

Equivalently, in Python interpreter:

```
>>> import sheepgame
>>> from sheepgame import sheepactions
>>> sheepactions.SheepJump()(n=5)
Sheep jumps 5 times

>>> sheepactions.SheepGraze()()
Sheep grazes on grass

>>> sheepactions.SheepGraze()(feed='daisies')
Sheep grazes on daisies
```

Wolf-related usage of `sheepgame.py`:

```
$ sheepgame.py feed-wolf -h
usage: sheepgame.py feed-wolf [-h] [-p PREY]

Feed the wolf

optional arguments:
  -h, --help  show this help message and exit
  -p PREY     Specify the food. Default:"sheep".
```

Examples of execution:

In the command line:

```
$ sheepgame.py feed-wolf
Wolf eats sheep
```

Equivalently, in Python interpreter

```
>>> import sheepgame
>>> sheepgame.FeedWolf()()
Wolf eats sheep
```

Tasks with multiple arguments

For simplicity, so far we have only considered functions of one argument. In practice, there is no such limitation.

For each argument of the function pointed to by the **`hook`** attribute there should be a call to **`add_argument()`** inside **`populate()`**, whose `dest=` value coincides with the name of the argument.

One should take full advantage of the rich set of options provided **`argparse`** methods such as **`add_argument()`**.

Here is an example, where the three arguments `quantity`, `feed`, and `hours` correspond to the three **`add_argument()`** calls with `dest='quantity'`, `dest='feed'` and `dest='hours'`:

```
import argpext

def sheep_graze(quantity,feed,hours):
    print( ('%s of sheep grazes on %s for %.1f hours.' \
              % (quantity, feed, hours) ).capitalize() )

class SheepGraze(argpext.Task):
    "Let sheep graze"
    hook = argpext.make_hook(sheep_graze)
    def populate(self,parser):
        parser.add_argument(dest='quantity', help='Quantity of sheep.')
        parser.add_argument('-f', dest='feed', default='grass',
                            help='Specify the feed. Default: %(default)s.')
        parser.add_argument('-t', dest='hours', default=2.5, type=float,
                            help='Specify number of hours. Default: %(default)s.')
```

```
if __name__ == '__main__':
    SheepGraze().digest()
```

The usage is as follows:

```
$ sheepgraze2.py -h
usage: sheepgraze2.py [-h] [-f FEED] [-t HOURS] quantity

Let sheep graze

positional arguments:
  quantity    Quantity of sheep.

optional arguments:
  -h, --help  show this help message and exit
  -f FEED     Specify the feed. Default: grass.
  -t HOURS    Specify number of hours. Default: 2.5.
```

Execution examples

In command line

```
$ sheepgraze2.py dosen
Dosen of sheep grazes on grass for 2.5 hours.

$ sheepgraze2.py herd -t 5
Herd of sheep grazes on grass for 5.0 hours.

$ sheepgraze2.py herd -f hay
Herd of sheep grazes on hay for 2.5 hours.
```

Equivalently, in Python interpreter

```
>>> import sheepgraze2
>>> sheepgraze2.SheepGraze()('dosen')
Dosen of sheep grazes on grass for 2.5 hours.

>>> sheepgraze2.SheepGraze()('herd',hours=5)
Herd of sheep grazes on grass for 5.0 hours.

>>> sheepgraze2.SheepGraze()('herd',feed='hay')
Herd of sheep grazes on hay for 2.5 hours.
```

Notice the agreement between the default values (e.g. `hour=2.5`) applied when an optional argument is missing in the command line examples and those in the corresponding Python interpreter examples.

Static `hook()` methods

Our *bare bones example* can be equivalently rewritten in a different style, as follows

```
import argpext
```

```
class SheepGraze(argpext.Task):
    "Let sheep graze"

    def hook(self,feed):
        print('Sheep grazes on %s' % feed)

    def populate(self,parser):
        parser.add_argument('-f', dest='feed', default='grass',
                            help='Specify the feed. Default: %(default)s.')

if __name__ == '__main__':
    SheepGraze().digest()
```

Return values

The **`Node.digest()`**, **`Task.digest()`** and **`Task.__call__()`** methods return the value of the corresponding reference function. For example:

```
>>> import argpext
>>>
... def square(x=1):
...     "Calculate the square of an argument"
...     return x*x
>>>
... class Square(argpext.Task):
...     hook = argpext.make_hook(square)
...     def populate(self,parser):
...         parser.add_argument('-x', default=2, type=float,
...                             help='Specify the value of x.')
>>>
>>> y = Square().digest(prog=None,args=['-x','2'])
>>> print( y )
4.0

>>>
>>> y = Square()(x=4)
>>> print( y )
16.0

>>>
>>> y = Square()()
>>> print( y )
4

>>>
>>> y = Square()() # Todo: add custom execution
>>> print( y )
4
```

Command line history log

Commands managed by Argpext are optionally saved into a local history. The feature is disabled by default; to enable it, set the environment variable **`ARGPEXT_HISTORY`** to specify the name of the history file.

Command line history is available by running **argpext.py** as executable with `history` sub-command.

KeyWords variable type

This section introduces class **`KeyWords`** to cover the type of variables whose possible values (or methods for generating those values) are known in advance; this is an alternative to using the `choices=` argument of **`argparse.add_argument()`**.

Consider the following possible mnemonic choices for specifying a date: “1977-02-04”, “Lisas birthday”, “y2kday”, “today”, and their implementation:

```
>>> import argpext
>>>
>>> from argpext import *
>>> import time
>>>
... def today():
...     "Return todays date in YYYY-MM-DD representation"
...     return time.strftime('%Y-%m-%d', time.localtime())
>>>
... dates = KeyWords([
...     '1977-02-04',
...     'Lisas birthday',
...     'y2kday',
...     'today'
>>> ])
>>>
>>>
>>>
>>> str(dates)
"KeyWords(['1977-02-04','Lisas birthday','y2kday','today'])"

>>>
>>> dates('1977-02-04')
'1977-02-04'

>>> dates('Lisas birthday')
'Lisas birthday'

>>> dates('y2kday')
'y2kday'

>>> dates('today') # Function today() is implicitly invoked at this line.
'today'

>>> dates('2012-01-11') # Value not predefined
KeyError: 'invalid key: "2012-01-11"'
```

The three predefined values of date are declared in lines 9-11; whereas line 12 declares a predefined method for finding the value of date:

Line 9: The value of the item is made identical to its reference key `1977-02-04`.
Line 10: The reference key is `Lisas birthday`; the value is fixed and equal to `1977-01-01`.

Line 11: The reference key is `y2kday`; the value is fixed and equal to `2000-01-01`.
Line 12: The reference key is `today`; the value is computed by function **`today()`** at the time of the actual evaluation (line 21).

Actual evaluations are shown in lines 18-27.

The **`KeyWords`** type object `dates`, constructed in the above example can be used as `type=` argument, similar to the case in our next *example*.

Treatment of unmatched values

The last evaluation (line 26) results in an error because the argument `2012-01-11` does not match any of predefined values.

The bare bones example revisited

Going back to *one* of our previously discussed examples **`KeyWords`** type values may be found particularly useful. Problems may arise because *command line usage* for that example allows one to pass any erroneous string as an argument. Indeed, consider this:

```
$ sheepgraze.py -f money
Sheep grazes on money
```

The **`KeyWords`** class allows one to limit the domain of argument values to a limited set of valid values and reflect the available choices in the usage. Introducing the **`KeyWords`** class into our example leads to the following:

```
import argpext

def sheep_graze(feed):
    print('Sheep grazes on %s' % feed)


class SheepGraze(argpext.Task):
    "Let sheep graze"

    hook = argpext.make_hook(sheep_graze)

    def populate(self,parser):
        parser.add_argument('-f', dest='feed', default='grass'
                            , type=argpext.KeyWords(['hay',
                                                     'grass',
                                                     'daisies'])
                            , help='Specify the feed. '\
                                'Choose from: %(type)s. '\
                                'Default: %(default)s.')

if __name__ == '__main__':
    SheepGraze().digest()
```

lines: 3-

emphasize-lines:

12-14,16

linenos:

The highlighted lines (12-14, and 16) emphasize changes relative to the *original program*.

After this modification, the valid values (`hay`, `grass`, and `daisies`) of input become visible within the help message. Indeed:

```
$ sheepgraze3.py -h
usage: sheepgraze3.py [-h] [-f FEED]

Let sheep graze

optional arguments:
  -h, --help  show this help message and exit
  -f FEED     Specify the feed. Choose from: 'hay', 'grass', and 'daisies'.
              Default: grass.
```

Examples of execution:

Passing any of the valid values results in proper execution:

```
$ sheepgraze3.py -f hay
Sheep grazes on hay

$ sheepgraze3.py -f daisies
Sheep grazes on daisies
```

Attempt to pass an erroneous argument leads to an error message:

```
$ sheepgraze3.py -f money
KeyError: 'invalid key: "money"'
```

Argpext as an executable

In addition to providing a Python module, program **argpext.py** can be ran as an executable; its current usage is as follows:

```
$ python -m argpext -h
usage: __main__.py [-h] {tasks,rst} ...

positional arguments:
  {tasks,rst}  Description
    tasks      Display command line history
    rst

optional arguments:
  -h, --help   show this help message and exit
```

Environment variables

ARGPEXT_HISTORY
Sub-command history file path. No history file is written if this variable is unset.

See also

- Argparse Sub-commands
- http://pypi.python.org/pypi/Baker
- https://github.com/anandology/subcommand

Contents:

Indices and tables

- *Index*
- *Module Index*
- *Search Page*