monolith Documentation

Release 0.3.3

Łukasz Balcerzak

December 16, 2013

Contents

Date December 16, 2013

Version 0.3.3

Documentation:

monolith is simple framwork for creating command line tools. Subcommands are class based (approach and part of implementation was inspired by Django management commands, however monolith uses *argparse* instead of *optparse*).

Supported Python versions are 2.6/2.7, 3.2+ and PyPy.

CHAPTER 1

Usage

Firstly, we need to build an entry point for our command line application. In `monolith` it is called *ExecutionManager*.

1.1 Execution manager

Create our execution manager

```
>>> from monolith.cli import ExecutionManager, BaseCommand
>>> manager = ExecutionManager()
```

1.2 Creating commands

Now let's create simple commands:

```
>>> class FooCommand(BaseCommand):
...     def handle(self, namespace):
...         print('foo', file=self.stdout)
...
>>> class BarCommand(BaseCommand):
...     def handle(self, namespace):
...         print('bar', file=self.stdout)
...
```

Note: Commands should write to specific stream explicitly or use *file* keyword argument of *print* function, but this would require to add following in Python 2.X:

```
from __future__ import print_function
```

1.3 Registering commands

Now register defined commands:

```
>>> manager.register('foo', FooCommand)
>>> manager.register('bar', BarCommand)
```

1.4 Commands execution

... and finally run them:

```
>>> manager.execute(['foo'])
foo
>>> manager.execute(['bar'])
bar
```

Note: Normally, in your program you would call *execute* method without any parameters - this would default to *sys.argv*.

1.5 Complete example

```
#!/usr/bin/env python
"""
Example of how to create git-like execution manager with monolith.
This is completely fake command.
"""
from __future__ import print_function
from __future__ import unicode_literals
from monolith.cli import ExecutionManager
from monolith.cli import LabelCommand
from monolith.cli import SingleLabelCommand
from monolith.cli import arg
from monolith.cli import CompletionCommand


class AddCommand(LabelCommand):

    def handle_label(self, label, namespace):
        print("A %s" % label, file=self.stdout)


class InitCommand(SingleLabelCommand):
    label = 'directory'
    label_required = False
    label_default_value = '.'
    args = SingleLabelCommand.args + [
        arg('--bare', help='Create a bare repository.', default=False,
            action='store_true'),
    ]

    def handle_label(self, label, namespace):
        print("Initialized empty Git repository in %s.git" % label,
            file=self.stdout)


def get_manager(**kwargs):
```

```
    manager = ExecutionManager(**kwargs)
    manager.register('add', AddCommand)
    manager.register('init', InitCommand)
    manager.register('completion', CompletionCommand),
    return manager

def main():
    manager = get_manager()
    manager.execute()


if __name__ == '__main__':
    main()
```

1.6 Simple execution manager

New in version 0.2. There is also possibility to use simple execution manager for more complex programs, i.e. if we create a package and put our commands in separate modules we can use *string to classes* instead of importing all command classes (you can still use imported commands too)

```
>>> manager = SimpleExecutionManager(program='foobar', commands={'sub-command': 'monolith.tests.test_
>>> manager.get_commands_to_register()
{'sub-command': <class 'monolith.tests.test_cli.DummyCommand'>, 'another-sub-command': <class 'monoli
```

CHAPTER 2

Installation

`monolith` runs on Python 2.6+/3.X. In order to install it simply use `easy_install`:

```
easy_install monolith
```

or `pip`:

```
pip install monolith
```

Note: As Python 2.6 was not yet shipped with `argparse` package, *distutils* would install it if installation is run under older Python version.

CHAPTER 3

Development

3.1 Testing

To run tests use nose:

```
$ nosetests
```

3.2 Tox

In order to run full test suite for all supported Python versions please use tox:

```
$ tox
```

3.3 Issues

Please file issues at https://github.com/lukaszb/monolith/issues. Also, if you fix something, please use *pull request* github's feature. Preferably, use separate branches per issue (in case there would be extra work needed, it's much easier to work locally at sepearate branch).

CHAPTER 4

License

CHAPTER 5

API Reference

5.1 monolith.cli

class monolith.cli.**Parser**(**args*, ***kwargs*)
Subclass of argparse.ArgumentParser providing more control over output stream.

5.1.1 SimpleExecutionManager

class monolith.cli.**SimpleExecutionManager**(*program*, *commands*)

Parameters

- **program** – name of the program under which commands would be executed (usually name of the program).
- **commands** – dictionary mapping subcommands to proper command classes. Values can be string - in that case proper command class would be importer and used. Example:

```
{
    'subcommand1': SomeCommand,
    'subcommand2': 'myprogram.commands.another.AnotherCommand',
}
```

get_commands_to_register()
Returns dictionary with commands given during construction. If value is a string, it would be converted into proper class pointer.

5.1.2 ExecutionManager

class monolith.cli.**ExecutionManager**(*argv=None*, *stderr=None*, *stdout=None*)

autocomplete()
If *completion* is enabled, this method would write to self.stdout completion words separated with space.

call_command(*cmd*, **argv*)
Runs a command.

Parameters

- **cmd** – command to run (key at the registry)
- **argv** – arguments that would be passed to the command

execute(*argv=None*)
Executes command based on given arguments.

get_commands()
Returns commands stored in the registry (sorted by name).

get_commands_to_register()
Returns dictionary (*name* / *Command* or string pointing at the command class.

get_parser()
Returns `monolith.cli.Parser` instance for this *ExecutionManager*.

get_usage()
Returns *usage* text of the main application parser.

parser_cls
alias of `Parser`

register(*name*, *Command*, *force=False*)
Registers given `Command` (as given `name`) at this *ExecutionManager*'s registry.

Parameters

- **name** – name in the registry under which given `Command` should be stored.
- **Command** – should be subclass of :class:`monolith.cli.base.BaseCommand`
- **force** – Forces registration if set to `True` - even if another command was already registered, it would be overridden and no execption would be raised. Defaults to `False`.

Raises AlreadyRegistered If another command was already registered under given `name`.

5.1.3 BaseCommand

class `monolith.cli.`**BaseCommand**(*prog_name=None*, *stdout=None*)
Base command class that should be subclassed by concrete commands.

Attributes

- `help`: Help description for this command. Defaults to empty string.
- `args`: List of `Argument` instances. Defaults to empty list.
- `prog_name`: Program name of *ExecutionManager* within which this command is run. Defaults to `None`.
- `stdout`: File-like object. Command should write to it. Defaults to `sys.stdout`.

get_args()
Returns list of `Argument` instances for the parser. By default, it returns `self.args`.

handle(*namespace*)
Handles given `namespace` and executes command. Should be overridden at subclass.

post_register(*manager*)
Performs actions once this command is registered within given `manager`. By default it does nothing.

setup_parser(*parser*, *cmdparser*)
This would be called when command is registered by ExecutionManager after arguments from `get_args` are processed.

Default implementation does nothing.

Parameters

- **parser** – Global argparser.ArgumentParser
- **cmdparser** – Subparser related with this command

5.1.4 LabelCommand

class `monolith.cli.`**LabelCommand**(*prog_name=None*, *stdout=None*)
Command that works on given position arguments (*labels*). By default, at least one *label* is required. This is controlled by *labels_required* attribute.

Extra attributes:

- `labels_required`: If `True`, at least one *label* is required, otherwise no positional arguments could be given. Defaults to `True`.

get_labels_arg()
Returns argument for *labels*.

handle(*namespace*)
Handles given `namespace` by calling `handle_label` method for each given *label*.

handle_label(*label*, *namespace*)
Handles single *label*. Should be overridden at subclass.

handle_no_labels(*namespace*)
Performs some action if no *lables* were given. By default it does nothing.

5.1.5 SingleLabelCommand

class `monolith.cli.`**SingleLabelCommand**(*prog_name=None*, *stdout=None*)
Command that works on given positional argument (*label*).

Extra arguments:

- `label_default_value`: If no *label* were given, this would be default value that would be passed to `namespace`. Defaults to `None`.

get_label_arg()
Returns argument for *label*.

handle(*namespace*)
Calls `handle_label` method for given *label*.

handle_label(*label*, *namespace*)
Handles *label*. Should be overridden at subclass.

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

m