Argon Documentation

Release 0.2.0

Simon Pantzare

October 08, 2012

CONTENTS

Helpers for sub-commands.

CHAPTER

ONE

INSTALLATION

```
$ pip install argon
```

CHAPTER

TWO

CONTENTS

2.1 Motivation

```
import sys
import argparse

import argon

def using_argparse(args):
    # create the top-level parser
    parser = argparse.ArgumentParser(prog="PROG")
    parser.add_argument("--foo", action="store_true", help="foo help")
    subparsers = parser.add_subparsers()

    # create the parser for the "a" command
    parser_a = subparsers.add_parser("a", help="a help")
    parser_a.add_argument("bar", type=int, help="bar help")
    parser_a.set_defaults(func=handle_a)

    # create the parser for the "b" command
    parser_b = subparsers.add_parser("b", help="b help")
    parser_b.add_argument("--baz", choices="XYZ", help="baz help")
    parser_b.set_defaults(func=handle_b)

    # parse arguments and run handler
    parsed = parser.parse_args(args)
    parsed.func(parsed)


def using_argon_with_validation(args):
    # create the top-level parser
    app = argon.App(prog="PROG")
    app.arg("--foo", action="store_true", help="foo help")

    # create the parser for the "a" command
    with app.command("a", help="a help") as a:
        a.arg("bar", type=int, help="bar help").handler(handle_a)

    # create the parser for the "b" command
    with app.command("b", help="b help") as b:
        b.arg("--baz", choices="XYZ", help="baz_help").handler(handle_b)

    # parse arguments and run handler
```

```
    app.run(args)


def using_argon_without_validation(args):
    # create the top-level parser
    app = argon.App(prog="PROG")
    app.arg("--foo", action="store_true", help="foo help")

    # create the parser for the "a" command
    app.command("a", help="a help") \
       .arg("bar", type=int, help="bar help") \
       .handler(handle_a)

    # create the parser for the "b" command
    app.command("b", help="b help") \
       .arg("--baz", choices="XYZ", help="baz_help") \
       .handler(handle_b)

    # parse arguments and run handler
    app.run(args)


def handle_a(args):
    print "handling a", args.foo, args.bar


def handle_b(args):
    print "handling b", args.foo, args.baz


if __name__ == "__main__":
    for using in [using_argparse, using_argon_with_validation,
                  using_argon_without_validation]:
        print "using", using
        for args in [["a", "12"], ["--foo", "b", "--baz", "Z"]]:
            using(args)
        print
```

2.2 Example

2.2.1 Usage

```
$ python example.py -h
usage: example.py [-h] [-r] {math,echo} ...

Argon example.

positional arguments:
  {math,echo}

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

Math Sub-Command Group

```
$ python example.py math -h
usage: example.py math [-h] {sum,hex} ...

positional arguments:
  {sum,hex}
    hex       convert number to hex

optional arguments:
  -h, --help  show this help message and exit
$ python example.py math sum 1 2 3
6
$ python example.py math hex 1234567
0x12d687
```

Echo Sub-Command

```
$ python example.py echo -h
usage: example.py echo [-h] [-u] string

positional arguments:
  string

optional arguments:
  -h, --help  show this help message and exit
  -u          convert to uppercase
$ python example.py echo fooo
fooo
$ python example.py -r echo -u fooo
OOOF
```

2.2.2 Source Code

```
import sys

import argon


def main():
    app = argon.App(description="Argon example.")

    # If given, the output is reversed.
    app.arg("-r", "--reverse", default=False, action="store_true",
            help="reverse output")

    # Add math sub-commands. "sum" computes the sum args and "hex" converts
    # an integer to hex.
    with app.sub("math") as math:
        with math.command("sum") as math_sum:
            math_sum.arg("numbers", help="numbers to compute sum of",
                         nargs="+", type=int)
            math_sum.handler(do_sum)
        with math.command("hex", help="convert number to hex") as math_hex:
            math_hex.arg("number", type=int).handler(do_hex)
```

```
    # Add an echo command. It accepts an argument to uppercase its input.
    with app.command("echo") as echo:
        echo.arg("-u", dest="uppercase", default=False, action="store_true",
                 help="convert to uppercase") \
            .arg("string") \
            .handler(do_echo)

    app.run(sys.argv[1:])


def show(func):
    """Decorator to print the return value of func, possibly reversed. """
    def wrapper(args):
        output = str(func(args))
        if args.reverse:
            print "".join(reversed(output))
        else:
            print output
    return wrapper


@show
def do_sum(args):
    return sum(args.numbers)


@show
def do_hex(args):
    return hex(args.number)


@show
def do_echo(args):
    if args.uppercase:
        return args.string.upper()
    return args.string


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

2.3 API Reference

class `argon.`**App**(**args*, ***kwds*)

Represents command-line applications.

Arguments are passed to `argparse.ArgumentParser`.

arg(**args*, ***kwds*)

Add an argument.

Arguments are passed to `argparse.ArgumentParser.add_argument()`.

command(*name*, **args*, ***kwds*)

Add a command.

This method adds a new parser to its subparsers. Arguments are passed to the `argparse.ArgumentParser` constructor.

parse (*args*)
Parse arguments.

Returns a two-tuple `(callable, parsed_args)`

parse_known (*args*)
Parse known arguments.

Returns a three-tuple `(callable, parsed_args, unknowns)`

run (*args*)
Run application. Typically `app.run(sys.argv[1:])`.

Same as:

```
>>> func, parsed_args = self.parse()
>>> func(parsed_args)
```

run_known (*args*)
Run application, allow unknown args.

Same as:

```
>>> func, parsed_args, unknowns = self.parse_known()
>>> func(parsed_args, unknowns)
```

sub (*name*, **args*, ***kwds*)
Add a sub-command group.

This method adds a new parser to its subparsers. Arguments are passed to the `argparse.ArgumentParser` constructor.

class `argon.`**Sub** (*parser*)
Used for sub-command groups.

arg (**args*, ***kwds*)
Add an argument.

Arguments are passed to `argparse.ArgumentParser.add_argument()`.

command (*name*, **args*, ***kwds*)
Add a command.

This method adds a new parser to its subparsers. Arguments are passed to the `argparse.ArgumentParser` constructor.

sub (*name*, **args*, ***kwds*)
Add a sub-command group.

This method adds a new parser to its subparsers. Arguments are passed to the `argparse.ArgumentParser` constructor.

class `argon.`**Command** (*parser*)
Used to create commands.

arg (**args*, ***kwds*)
Add an argument.

Arguments are passed to `argparse.ArgumentParser.add_argument()`.

handler (*func*)
Set handler function for command.

The function must accept a single argument *args* (output from `argparse.ArgumentParser.parse_args()`).

2.4 Tests

```
import unittest

import argon


def returns(return_value):
    return lambda args: return_value


class TestArgon(unittest.TestCase):

    def test_command_ok(self):
        app = argon.App()
        with app.command("a") as a:
            a.handler(returns("a"))
        self.assertEqual(app.run(["a"]), "a")

    def test_command_no_handler(self):
        def no_handler():
            app = argon.App()
            with app.command("a") as a:
                pass  # no handler
        self.assertRaises(argon.CommandError, no_handler)

    def test_command_two_handlers(self):
        def two_handlers():
            app = argon.App()
            with app.command("a") as a:
                a.handler(returns("a"))
                a.handler(returns("a"))
        self.assertRaises(argon.CommandError, two_handlers)

    def test_sub_simple_ok(self):
        app = argon.App()
        with app.sub("a") as a:
            with a.command("a") as aa:
                aa.handler(returns("aa"))
            with a.command("b") as ab:
                ab.handler(returns("ab"))
        with app.command("c") as a:
            a.handler(returns("c"))
        self.assertEqual(app.run(["a", "a"]), "aa")
        self.assertEqual(app.run(["a", "b"]), "ab")
        self.assertEqual(app.run(["c"]), "c")

    def test_nesting(self):
        app = argon.App()
        with app.sub("a") as a:
            with a.sub("a") as aa:
```

```
                with aa.command("a") as aaa:
                    aaa.handler(returns("aaa"))
                with aa.command("b") as aab:
                    aab.handler(returns("aab"))
                with aa.sub("c") as aac:
                    with aac.command("a") as aaca:
                        aaca.handler(returns("aaca"))
                    with aac.command("b") as aacb:
                        aacb.handler(returns("aacb"))
            with a.sub("b") as ab:
                with ab.command("a") as aba:
                    aba.handler(returns("aba"))
                with ab.command("b") as abb:
                    abb.handler(returns("abb"))
                with ab.sub("c") as abc:
                    with abc.command("a") as abca:
                        abca.handler(returns("abca"))
                    with abc.command("b") as abcb:
                        abcb.handler(returns("abcb"))

        for comb in ["aaa", "aab", "aaca", "aacb",
                     "aba", "abb", "abca", "abcb"]:
            self.assertEqual(app.run(list(comb)), comb)

    def test_name_collisions(self):
        def same_sub_sub_name():
            app = argon.App()
            with app.sub("a") as a:
                with a.command("aa") as aa:
                    aa.handler(returns("aa"))
            with app.sub("a") as a:
                with a.command("aa") as aa:
                    aa.handler(returns("aa"))
        self.assertRaises(argon.SubError, same_sub_sub_name)

        def same_sub_command_name():
            app = argon.App()
            with app.sub("a") as a:
                with a.command("aa") as aa:
                    aa.handler(returns("aa"))
            with app.command("a") as a:
                a.handler(returns("a"))
        self.assertRaises(argon.SubError, same_sub_command_name)

        def same_command_command_name():
            app = argon.App()
            with app.command("a") as a:
                a.handler(returns("a"))
            with app.command("a") as a:
                a.handler(returns("a"))
        self.assertRaises(argon.SubError, same_command_command_name)

    def test_args(self):
        def xy(args):
            return (args.x, args.y)

        app = argon.App()
        with app.sub("a") as a:
```

```
            a.arg("-x", dest="x", nargs="?", default=False)
            with a.command("a") as aa:
                aa.arg("-y", dest="y", default=False, action="store_true")
                aa.handler(xy)

        self.assertEqual(app.run("a a".split()), (False, False))
        self.assertEqual(app.run("a -x a a".split()), ("a", False))
        self.assertEqual(app.run("a a -y".split()), (False, True))
        self.assertEqual(app.run("a -x foo a -y".split()), ("foo", True))

    def test_parse(self):
        handler = lambda args: args
        app = argon.App()
        with app.command("a") as a:
            a.handler(handler)
        func, args = app.parse(["a"])
        self.assertIs(func, handler)

    def test_parse_known(self):
        handler = lambda args: args
        app = argon.App()
        with app.command("a") as a:
            a.handler(handler)
        func, args, unknowns = app.parse_known(["a", "b"])
        self.assertIs(func, handler)
        self.assertEqual(unknowns, ["b"])

    def test_run_known(self):
        return_unknowns = lambda args, unknowns: unknowns
        app = argon.App()
        with app.command("a") as a:
            a.handler(return_unknowns)
        self.assertEqual(app.run_known(["a", "b"]), ["b"])

if __name__ == "__main__":
    unittest.main()
```

CHAPTER

THREE

INDICES AND TABLES

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

PYTHON MODULE INDEX

a