argh Documentation

Release 0.26.3-dev

Andrey Mikhaylenko

September 05, 2016

Contents

Building a command-line interface? Found yourself uttering "argh!" while struggling with the API of *argparse*? Don't like the complexity but need the power?

> Everything should be made as simple as possible, but no simpler.
>
> —Albert Einstein (probably)

Argh is a smart wrapper for *argparse*. *Argparse* is a very powerful tool; *Argh* just makes it easy to use.

CHAPTER 1

In a nutshell

Argh-powered applications are *simple* but *flexible*:

Modular Declaration of commands can be decoupled from assembling and dispatching;

Pythonic Commands are declared naturally, no complex API calls in most cases;

Reusable Commands are plain functions, can be used directly outside of CLI context;

Layered The complexity of code raises with requirements;

Transparent The full power of argparse is available whenever needed;

Namespaced Nested commands are a piece of cake, no messing with subparsers (though they are of course used under the hood);

Term-Friendly Command output is processed with respect to stream encoding;

Unobtrusive *Argh* can dispatch a subset of pure-*argparse* code, and pure-*argparse* code can update and dispatch a parser assembled with *Argh*;

DRY The amount of boilerplate code is minimal; among other things, *Argh* will:

- infer command name from function name;
- infer arguments from function signature;
- infer argument type from the default value;
- infer argument action from the default value (for booleans);
- add an alias root command `help` for the `--help` argument.

NIH free *Argh* supports *completion*, *progress bars* and everything else by being friendly to excellent 3rd-party libraries. No need to reinvent the wheel.

Sounds good? Check the tutorial!

CHAPTER 2

Relation to argparse

Argh is fully compatible with *argparse*. You can mix *Argh*-agnostic and *Argh*-aware code. Just keep in mind that the dispatcher does some extra work that a custom dispatcher may not do.

CHAPTER 3

Installation

Using pip:

```
$ pip install argh
```

Arch Linux (AUR):

```
$ yaourt python-argh
```

CHAPTER 4

Examples

A very simple application with one command:

```
import argh

def main():
    return 'Hello world'

argh.dispatch_command(main)
```

Run it:

```
$ ./app.py
Hello world
```

A potentially modular application with multiple commands:

```
import argh

# declaring:

def echo(text):
    "Returns given word as is."
    return text

def greet(name, greeting='Hello'):
    "Greets the user with given name. The greeting is customizable."
    return greeting + ', ' + name

# assembling:

parser = argh.ArghParser()
parser.add_commands([echo, greet])

# dispatching:

if __name__ == '__main__':
    parser.dispatch()
```

Of course it works:

```
$ ./app.py greet Andy
Hello, Andy
```

```
$ ./app.py greet Andy -g Arrrgh
Arrrgh, Andy
```

Here's the auto-generated help for this application (note how the docstrings are reused):

```
$ ./app.py help

usage: app.py {echo,greet} ...

positional arguments:
    echo        Returns given word as is.
    greet       Greets the user with given name. The greeting is customizable.
```

...and for a specific command (an ordinary function signature is converted to CLI arguments):

```
$ ./app.py help greet

usage: app.py greet [-g GREETING] name

Greets the user with given name. The greeting is customizable.

positional arguments:
  name

optional arguments:
  -g GREETING, --greeting GREETING   'Hello'
```

(The help messages have been simplified a bit for brevity.)

Argh easily maps plain Python functions to CLI. Sometimes this is not enough; in these cases the powerful API of *argparse* is also available:

```
@arg('text', default='hello world', nargs='+', help='The message')
def echo(text):
    print text
```

The approaches can be safely combined even up to this level:

```
# adding help to `foo` which is in the function signature:
@arg('foo', help='blah')
# these are not in the signature so they go to **kwargs:
@arg('baz')
@arg('-q', '--quux')
# the function itself:
def cmd(foo, bar=1, *args, **kwargs):
    yield foo
    yield bar
    yield ', '.join(args)
    yield kwargs['baz']
    yield kwargs['quux']
```

CHAPTER 5

Links

- Project home page (GitHub)
- Documentation (Read the Docs)
- Package distribution (PyPI)
- Questions, requests, bug reports, etc.:
  - Issue tracker (GitHub)
  - Mailing list (subscribe to get important announcements)
  - Direct e-mail (neithere at gmail com)

CHAPTER 6

Author

Developed by Andrey Mikhaylenko since 2010.

See file *AUTHORS* for a complete list of contributors to this library.

CHAPTER 7

Support

The fastest way to improve this project is to submit tested and documented patches or detailed bug reports.

Otherwise you can “flattr” me:

CHAPTER 8

Licensing

Argh is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Argh is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Argh. If not, see <http://gnu.org/licenses/>.

CHAPTER 9

Dependencies

The *argh* library is supported (and tested unless otherwise specified) on the following versions of Python:

- 2.7 (including PyPy 1.8)
- 3.1 (*argparse* library is required; **not** tested)
- 3.4
- 3.5

Changed in version 0.15: Added support for Python 3.x, dropped support for Python 2.5.

Changed in version 0.18: Improved support for Python 3.2, added support for Python 3.3.

Changed in version 0.25: Added support for Python 3.4, dropped support for Python 3.3. Argh may perfectly work under 3.3, I'm just not testing it.

Changed in version 0.27: Added support for Python 3.5, dropped support for Python 2.6 and 3.2.

CHAPTER 10

Why this one?

See Similar projects.

CHAPTER 11

Details

11.1 Tutorial

Argh is a small library that provides several layers of abstraction on top of *argparse*. You are free to use any layer that fits given task best. The layers can be mixed. It is always possible to declare a command with the highest possible (and least flexible) layer and then tune the behaviour with any of the lower layers including the native API of *argparse*.

11.1.1 Dive In

Assume we need a CLI application which output is modulated by arguments:

```
$ ./greet.py
Hello unknown user!

$ ./greet.py --name John
Hello John!
```

This is our business logic:

```
def main(name='unknown user'):
    return 'Hello {0}!'.format(name)
```

That was plain Python, nothing CLI-specific. Let's convert the function into a complete CLI application:

```
argh.dispatch_command(main)
```

Done. Dead simple.

What about multiple commands? Easy:

```
argh.dispatch_commands([load, dump])
```

And then call your script like this:

```
$ ./app.py dump
$ ./app.py load fixture.json
$ ./app.py load fixture.yaml --format=yaml
```

I guess you get the picture. The commands are **ordinary functions** with ordinary signatures:

- Declare them somewhere, dispatch them elsewhere. This ensures **loose coupling** of components in your application.

- They are **natural** and pythonic. No fiddling with the parser and the related intricacies like `action='store_true'` which you could never remember.

Still, there's much more to commands than this.

The examples above raise some questions, including:

- do we have to `return`, or `print` and `yield` are also supported?
- what's the difference between `dispatch_command()` and `dispatch_commands()`? What's going on under the hood?
- how do I add help for each argument?
- how do I access the parser to fine-tune its behaviour?
- how to keep the code as DRY as possible?
- how do I expose the function under custom name and/or define aliases?
- how do I have values converted to given type?
- can I use a namespace object instead of the natural way?

Just read on.

11.1.2 Declaring Commands

The Natural Way

You've already learned the natural way of declaring commands before even knowing about *argh*:

```
def my_command(alpha, beta=1, gamma=False, *delta):
    return
```

When executed as `app.py my-command --help`, such application prints:

```
usage: app.py my-command [-h] [-b BETA] [-g] alpha [delta [delta ...]]

positional arguments:
  alpha
  delta

optional arguments:
  -h, --help            show this help message and exit
  -b BETA, --beta BETA
  -g, --gamma
```

The same result can be achieved with this chunk of *argparse* code (with the exception that in *argh* you don't immediately modify a parser but rather declare what's to be added to it later):

```
parser.add_argument('alpha')
parser.add_argument('-b', '--beta', default=1, type=int)
parser.add_argument('-g', '--gamma', default=False, action='store_true')
parser.add_argument('delta', nargs='*')
```

Verbose, hardly readable, requires learning another API.

Argh allows for more expressive and pythonic code because:

- everything is inferred from the function signature;
- arguments without default values are interpreted as required positional arguments;

- arguments with default values are interpreted as options;
  - options with a *bool* as default value are considered flags and their presence triggers the action *store_true* (or *store_false*);
  - values of options that don't trigger actions are coerced to the same type as the default value;
- the `*args` entry (function's positional arguments) is interpreted as a single argument with 0..n values.

Hey, that's a lot for such a simple case! But then, that's why the API feels natural: *argh* does a lot of work for you.

Well, there's nothing more elegant than a simple function. But simplicity comes at a cost in terms of flexibility. Fortunately, *argh* doesn't stay in the way and offers less natural but more powerful tools.

Documenting Your Commands

The function's docstring is automatically included in the help message. When the script is called as `./app.py my-command --help`, the docstring is displayed along with a short overview of the arguments.

However, in many cases it's a good idea do add extra documentation per argument.

In Python 3 it's easy:

```
def load(path : 'file to load', format : 'json or yaml' = 'yaml'):
    "Loads given file as YAML (unless other format is specified)"
    return loaders[format].load(path)
```

Python 2 does not support annotations so the above example would raise a *SyntaxError*. You would need to add help via *argparse* API:

```
parser.add_argument('path', help='file to load')
```

...which is far from DRY and very impractical if the functions are dispatched in a different place. This is when extended declarations become useful.

Extended Argument Declaration

When function signature isn't enough to fine-tune the argument declarations, the *arg* decorator comes in handy:

```
@arg('path', help='file to load')
@arg('--format', help='json or yaml')
def load(path, format='yaml'):
    return loaders[format].load(path)
```

In this example we have declared a function with arguments *path* and *format* and then extended their declarations with help messages.

The decorator mostly mimics *argparse*'s add_argument. The *name_or_flags* argument must match function signature, that is:

1. `path` and `--format` map to `func(path)` and `func(format='x')` respectively (short name like `-f` can be omitted);
2. a name that doesn't map to anything in function signature is not allowed.

The decorator doesn't modify the function's behaviour in any way.

Sometimes the function is not likely to be used other than as a CLI command and all of its arguments are duplicated with decorators. Not very DRY. In this case `**kwargs` can be used as follows:

```
@arg('number', default=0, help='the number to increment')
def increment(**kwargs):
    return kwargs['number'] + 1
```

In other words, if `**something` is in the function signature, extra arguments are **allowed** to be specified via decorators; they all go into that very dictionary.

Mixing `**kwargs` with straightforward signatures is also possible:

```
@arg('--bingo')
def cmd(foo, bar=1, *maybe, **extra):
    return ...
```

Note: It is not recommended to mix `*args` with extra *positional* arguments declared via decorators because the results can be pretty confusing (though predictable). See *argh* tests for details.

Namespace Objects

The default approach of *argparse* is similar to `**kwargs`: the function expects a single object and the CLI arguments are defined elsewhere.

In order to dispatch such "argparse-style" command via *argh*, you need to tell the latter that the function expects a namespace object. This is done by wrapping the function into the *expects_obj()* decorator:

```
@expects_obj
def cmd(args):
    return args.foo
```

This way arguments cannot be defined in the Natural Way but the *arg* decorator works as usual.

Note: In both cases — `**kwargs`-only and *@expects_obj* — the arguments **must** be declared via decorators or directly via the *argparse* API. Otherwise the command has zero arguments (apart from `--help`).

11.1.3 Assembling Commands

Note: *Argh* decorators introduce a declarative mode for defining commands. You can access the *argparse* API after a parser instance is created.

After the commands are declared, they should be assembled within a single argument parser. First, create the parser itself:

```
parser = argparse.ArgumentParser()
```

Add a couple of commands via *add_commands()*:

```
argh.add_commands(parser, [load, dump])
```

The commands will be accessible under the related functions' names:

```
$ ./app.py {load,dump}
```

Subcommands

If the application has too many commands, they can be grouped into namespaces:

```
argh.add_commands(parser, [serve, ping], namespace='www',
                  title='Web-related commands')
```

The resulting CLI is as follows:

```
$ ./app.py www {serve,ping}
```

See Subparsers for the gory details.

11.1.4 Dispatching Commands

The last thing is to actually parse the arguments and call the relevant command (function) when our module is called as a script:

```
if __name__ == '__main__':
    argh.dispatch(parser)
```

The function *dispatch()* uses the parser to obtain the relevant function and arguments; then it converts arguments to a form digestible by this particular function and calls it. The errors are wrapped if required (see below); the output is processed and written to *stdout* or a given file object. Special care is given to terminal encoding. All this can be fine-tuned, see API docs.

A set of commands can be assembled and dispatched at once with a shortcut *dispatch_commands()* which isn't as flexible as the full version described above but helps reduce the code in many cases. Please refer to the API documentation for details.

Modular Application

As you can see, with *argh* the CLI application consists of three parts:

1. declarations (functions and their arguments);
2. assembling (a parser is constructed with these functions);
3. dispatching (input → parser → function → output).

This clear separation makes a simple script just a bit more readable, but for a large application this is extremely important.

Also note that the parser is standard. It's OK to call *dispatch()* on a custom subclass of *argparse.ArgumentParser*.

By the way, *argh* ships with *ArghParser* which integrates the assembling and dispatching functions for DRYness.

Entry Points

New in version 0.25.

The normal way is to declare commands, then assemble them into an entry point and then dispatch.

However, It is also possible to first declare an entry point and then register the commands with it right at command declaration stage.

The commands are assembled together but the parser is not created until dispatching.

To do so, use *EntryPoint*:

```
from argh import EntryPoint


app = EntryPoint('my cool app')

@app
def foo():
    return 'hello'

@app
def bar():
    return 'bye'


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

11.1.5 Single-command application

There are cases when the application performs a single task and it perfectly maps to a single command. The method above would require the user to type a command like `check_mail.py check --now` while `check_mail.py --now` would suffice. In such cases *`add_commands()`* should be replaced with *`set_default_command()`*:

```
def main():
    return 1

argh.set_default_command(parser, main)
```

There's also a nice shortcut *`dispatch_command()`*. Please refer to the API documentation for details.

11.1.6 Subcommands + Default Command

New in version 0.26.

It's possible to augment a single-command application with nested commands:

```
p = ArghParser()
p.add_commands([foo, bar])
p.set_default_command(foo)     # could be a `quux`
```

However, this will raise an exception on assembling stage unless you have at least Python 3.4. The reason is a bug in *argparse*. Alright, what should you do then? This is a simple workaround:

```
p = argh.ArghParser()
p.add_commands([foo, bar])
try:
    p.set_default_command(quux)
except argh.AssemblingError:
    print('Please upgrade to Python 3.4 or higher')
    p.add_commands([quux])
```

Note: If you are using *argh* with barebones *ArgumentParser*, make sure that the *parse_args()* method gets `ArghNamespace` as the namespace object, otherwise the correct choice of function cannot be guaranteed. The

reason is that a higher-level parser has higher priority than its nested ones when *argparse* picks a default dest from defaults, which includes the function mapped to a certain endpoint.

11.1.7 Generated help

Argparse takes care of generating nicely formatted help for commands and arguments. The usage information is displayed when user provides the switch `--help`. However *argparse* does not provide a `help` *command*.

Argh always adds the command `help` automatically:

- `help shell` → `shell --help`
- `help web serve` → `web serve --help`

See also *#documenting-your-commands*.

11.1.8 Returning results

Most commands print something. The traditional straightforward way is this:

```
def foo():
    print('hello')
    print('world')
```

However, this approach has a couple of flaws:

- it is difficult to test functions that print results: you are bound to doctests or need to mess with replacing stdout;
- terminals and pipes frequently have different requirements for encoding, so Unicode output may break the pipe (e.g. `$ foo.py test | wc -l`). Of course you don't want to do the checks on every *print* statement.

Good news: if you return a string, *Argh* will take care of the encoding:

```
def foo():
    return ''
```

But what about multiple print statements? Collecting the output in a list and bulk-processing it at the end would suffice. Actually you can simply return a list and *Argh* will take care of it:

```
def foo():
    return ['hello', 'world']
```

Note: If you return a string, it is printed as is. A list or tuple is iterated and printed line by line. This is how *dispatcher* works.

This is fine, but what about non-linear code with if/else, exceptions and interactive prompts? Well, you don't need to manage the stack of results within the function. Just convert it to a generator and *Argh* will do the rest:

```
def foo():
    yield 'hello'
    yield 'world'
```

Syntactically this is exactly the same as the first example, only with *yield* instead of *print*. But the function becomes much more flexible.

Hint: If your command is likely to output Unicode and be used in pipes, you should definitely use the last approach.

11.1.9 Exceptions

Usually you only want to display the traceback on unexpected exceptions. If you know that something can be wrong, you'll probably handle it this way:

```
def show_item(key):
    try:
        item = items[key]
    except KeyError as error:
        print(e)     # hide the traceback
        sys.exit()   # bail out (unsafe!)
    else:
        ... do something ...
        print(item)
```

This works, but the print-and-exit tasks are repetitive; moreover, there are cases when you don't want to raise *SystemExit* and just need to collect the output in a uniform way. Use *CommandError*:

```
def show_item(key):
    try:
        item = items[key]
    except KeyError as error:
        raise CommandError(error)  # bail out, hide traceback
    else:
        ... do something ...
        return item
```

Argh will wrap this exception and choose the right way to display its message (depending on how *dispatch()* was called).

Decorator *wrap_errors()* reduces the code even further:

```
@wrap_errors([KeyError])  # catch KeyError, show the message, hide traceback
def show_item(key):
    return items[key]     # raise KeyError
```

Of course it should be used with care in more complex commands.

The decorator accepts a list as its first argument, so multiple commands can be specified. It also allows plugging in a preprocessor for the caught errors:

```
@wrap_errors(processor=lambda excinfo: 'ERR: {0}'.format(excinfo))
def func():
    raise CommandError('some error')
```

The command above will print *ERR: some error*.

11.1.10 Packaging

So, you've done with the first version of your *Argh*-powered app. The next step is to package it for distribution. How to tell *setuptools* to create a system-wide script? A simple example sums it up:

```
from setuptools import setup, find_packages

setup(
    name = 'myapp',
    version = '0.1',
    entry_points = {'console_scripts': ['myapp = myapp:main']},
    packages = find_packages(),
    install_requires = ['argh'],
)
```

This creates a system-wide *myapp* script that imports the *myapp* module and calls a *myapp.main* function.

More complex examples can be found in this contributed repository: https://github.com/illumin-us-r3v0lution/argh-examples

11.2 API Reference

11.2.1 Command decorators

argh.decorators.**aliases**(**names*)

Defines alternative command name(s) for given function (along with its original name). Usage:

```
    @aliases('co', 'check')
    def checkout(args):
        ...
```

The resulting command will be available as `checkout`, `check` and `co`.

Note: This decorator only works with a recent version of argparse (see Python issue 9324 and Python rev 4c0426). Such version ships with **Python 3.2+** and may be available in other environments as a separate package. Argh does not issue warnings and simply ignores aliases if they are not supported. See *SUPPORTS_ALIASES*.

New in version 0.19.

argh.decorators.**named**(*new_name*)

Sets given string as command name instead of the function name. The string is used verbatim without further processing.

Usage:

```
    @named('load')
    def do_load_some_stuff_and_keep_the_original_function_name(args):
        ...
```

The resulting command will be available only as `load`. To add aliases without renaming the command, check *aliases()*.

New in version 0.19.

argh.decorators.**arg**(**args*, ***kwargs*)

Declares an argument for given function. Does not register the function anywhere, nor does it modify the function in any way.

The signature of the decorator matches that of `argparse.ArgumentParser.add_argument()`, only some keywords are not required if they can be easily guessed (e.g. you don't have to specify type or action when an *int* or *bool* default value is supplied).

Typical use cases:

- In combination with *expects_obj()* (which is not recommended);
- in combination with ordinary function signatures to add details that cannot be expressed with that syntax (e.g. help message).

Usage:

```
from argh import arg

@arg('path', help='path to the file to load')
@arg('--format', choices=['yaml','json'])
@arg('-v', '--verbosity', choices=range(0,3), default=2)
def load(path, something=None, format='json', dry_run=False, verbosity=1):
    loaders = {'json': json.load, 'yaml': yaml.load}
    loader = loaders[args.format]
    data = loader(args.path)
    if not args.dry_run:
        if verbosity < 1:
            print('saving to the database')
        put_to_database(data)
```

In this example:

- *path* declaration is extended with *help*;
- *format* declaration is extended with *choices*;
- *dry_run* declaration is not duplicated;
- *verbosity* is extended with *choices* and the default value is overridden. (If both function signature and @*arg* define a default value for an argument, @*arg* wins.)

Note: It is recommended to avoid using this decorator unless there's no way to tune the argument's behaviour or presentation using ordinary function signatures. Readability counts, don't repeat yourself.

`argh.decorators.`**`wrap_errors`**(*errors=None*, *processor=None*, **args*)

Decorator. Wraps given exceptions into *CommandError*. Usage:

```
@wrap_errors([AssertionError])
def foo(x=None, y=None):
    assert x or y, 'x or y must be specified'
```

If the assertion fails, its message will be correctly printed and the stack hidden. This helps to avoid boilerplate code.

Parameters

- **`errors`** – A list of exception classes to catch.
- **`processor`** – A callable that expects the exception object and returns a string. For example, this renders all wrapped errors in red colour:

```
from termcolor import colored

def failure(err):
```

```
        return colored(str(err), 'red')

    @wrap_errors(processor=failure)
    def my_command(...):
        ...
```

argh.decorators.**expects_obj**(*func*)
Marks given function as expecting a namespace object.

Usage:

```
@arg('bar')
@arg('--quux', default=123)
@expects_obj
def foo(args):
    yield args.bar, args.quux
```

This is equivalent to:

```
def foo(bar, quux=123):
    yield bar, quux
```

In most cases you don't need this decorator.

11.2.2 Assembling

Functions and classes to properly assemble your commands in a parser.

argh.assembling.**SUPPORTS_ALIASES = False**
Calculated on load. If *True*, current version of argparse supports alternative command names (can be set via *aliases()*).

argh.assembling.**set_default_command**(*parser*, *function*)
Sets default command (i.e. a function) for given parser.

If *parser.description* is empty and the function has a docstring, it is used as the description.

Note: An attempt to set default command to a parser which already has subparsers (e.g. added with *add_commands()*) results in a *AssemblingError*.

Note: If there are both explicitly declared arguments (e.g. via *arg()*) and ones inferred from the function signature (e.g. via `command()`), declared ones will be merged into inferred ones. If an argument does not conform function signature, *AssemblingError* is raised.

Note: If the parser was created with `add_help=True` (which is by default), option name `-h` is silently removed from any argument.

---

argh.assembling.**add_commands**(*parser*, *functions*, *namespace=None*, *namespace_kwargs=None*, *func_kwargs=None*, *title=None*, *description=None*, *help=None*)
Adds given functions as commands to given parser.

Parameters

- **parser** – an `argparse.ArgumentParser` instance.

- **functions** – a list of functions. A subparser is created for each of them. If the function is decorated with *arg()*, the arguments are passed to `argparse.ArgumentParser.add_argument`. See also *dispatch()* for requirements concerning function signatures. The command name is inferred from the function name. Note that the underscores in the name are replaced with hyphens, i.e. function name "foo_bar" becomes command name "foo-bar".
- **namespace** – an optional string representing the group of commands. For example, if a command named "hello" is added without the namespace, it will be available as "prog.py hello"; if the namespace if specified as "greet", then the command will be accessible as "prog.py greet hello". The namespace itself is not callable, so "prog.py greet" will fail and only display a help message.
- **func_kwargs** – a *dict* of keyword arguments to be passed to each nested ArgumentParser instance created per command (i.e. per function). Members of this dictionary have the highest priority, so a function's docstring is overridden by a *help* in *func_kwargs* (if present).
- **namespace_kwargs** – a *dict* of keyword arguments to be passed to the nested ArgumentParser instance under given *namespace*.

Deprecated params that should be moved into *namespace_kwargs*:

Parameters

- **title** – passed to `argparse.ArgumentParser.add_subparsers()` as *title*.

  Deprecated since version 0.26.0: Please use *namespace_kwargs* instead.
- **description** – passed to `argparse.ArgumentParser.add_subparsers()` as *description*.

  Deprecated since version 0.26.0: Please use *namespace_kwargs* instead.
- **help** – passed to `argparse.ArgumentParser.add_subparsers()` as *help*.

  Deprecated since version 0.26.0: Please use *namespace_kwargs* instead.

Note: This function modifies the parser object. Generally side effects are bad practice but we don't seem to have any choice as ArgumentParser is pretty opaque. You may prefer *add_commands* for a bit more predictable API.

Note: An attempt to add commands to a parser which already has a default function (e.g. added with *set_default_command()*) results in *AssemblingError*.

`argh.assembling.`**add_subcommands**(*parser*, *namespace*, *functions*, ***namespace_kwargs*)

A wrapper for *add_commands()*.

These examples are equivalent:

```
add_commands(parser, [get, put], namespace='db',
             namespace_kwargs={
                 'title': 'database commands',
                 'help': 'CRUD for our silly database'
             })

add_subcommands(parser, 'db', [get, put],
                title='database commands',
                help='CRUD for our silly database')
```

11.2.3 Dispatching

argh.dispatching.**dispatch**(*parser*, *argv=None*, *add_help_command=True*, *completion=True*, *pre_call=None*, *output_file=<open file '<stdout>', mode 'w'>*, *errors_file=<open file '<stderr>', mode 'w'>*, *raw_output=False*, *namespace=None*, *skip_unknown_args=False*)

Parses given list of arguments using given parser, calls the relevant function and prints the result.

The target function should expect one positional argument: the `argparse.Namespace` object. However, if the function is decorated with `plain_signature()`, the positional and named arguments from the namespace object are passed to the function instead of the object itself.

Parameters

- **parser** – the ArgumentParser instance.
- **argv** – a list of strings representing the arguments. If *None*, `sys.argv` is used instead. Default is *None*.
- **add_help_command** – if *True*, converts first positional argument "help" to a keyword argument so that `help foo` becomes `foo --help` and displays usage information for "foo". Default is *True*.
- **output_file** – A file-like object for output. If *None*, the resulting lines are collected and returned as a string. Default is `sys.stdout`.
- **errors_file** – Same as *output_file* but for `sys.stderr`.
- **raw_output** – If *True*, results are written to the output file raw, without adding whitespaces or newlines between yielded strings. Default is *False*.
- **completion** – If *True*, shell tab completion is enabled. Default is *True*. (You will also need to install it.) See *argh.completion*.
- **skip_unknown_args** – If *True*, unknown arguments do not cause an error (*ArgumentParser.parse_known_args* is used).
- **namespace** – An *argparse.Namespace*-like object. By default an `ArghNamespace` object is used. Please note that support for combined default and nested functions may be broken if a different type of object is forced.

By default the exceptions are not wrapped and will propagate. The only exception that is always wrapped is *CommandError* which is interpreted as an expected event so the traceback is hidden. You can also mark arbitrary exceptions as "wrappable" by using the *wrap_errors()* decorator.

argh.dispatching.**dispatch_command**(*function*, **args*, ***kwargs*)

A wrapper for *dispatch()* that creates a one-command parser. Uses `PARSER_FORMATTER`.

This:

```
dispatch_command(foo)
```

...is a shortcut for:

```
parser = ArgumentParser()
set_default_command(parser, foo)
dispatch(parser)
```

This function can be also used as a decorator.

argh.dispatching.**dispatch_commands**(*functions*, **args*, ***kwargs*)

A wrapper for *dispatch()* that creates a parser, adds commands to the parser and dispatches them. Uses `PARSER_FORMATTER`.

This:

```
dispatch_commands([foo, bar])
```

...is a shortcut for:

```
parser = ArgumentParser()
add_commands(parser, [foo, bar])
dispatch(parser)
```

class argh.dispatching.**EntryPoint**(*name=None*, *parser_kwargs=None*)
An object to which functions can be attached and then dispatched.

When called with an argument, the argument (a function) is registered at this entry point as a command.

When called without an argument, dispatching is triggered with all previously registered commands.

Usage:

```
from argh import EntryPoint

app = EntryPoint('main', dict(description='This is a cool app'))

@app
def ls():
    for i in range(10):
        print i

@app
def greet():
    print 'hello'

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

11.2.4 Interaction

argh.interaction.**confirm**(*action*, *default=None*, *skip=False*)
A shortcut for typical confirmation prompt.

Parameters

- **action** – a string describing the action, e.g. "Apply changes". A question mark will be appended.
- **default** – *bool* or *None*. Determines what happens when user hits Enter without typing in a choice. If *True*, default choice is "yes". If *False*, it is "no". If *None* the prompt keeps reappearing until user types in a choice (not necessarily acceptable) or until the number of iteration reaches the limit. Default is *None*.
- **skip** – *bool*; if *True*, no interactive prompt is used and default choice is returned (useful for batch mode). Default is *False*.

Usage:

```
def delete(key, silent=False):
    item = db.get(Item, args.key)
    if confirm('Delete '+item.title, default=True, skip=silent):
        item.delete()
        print('Item deleted.')
```

```
        else:
            print('Operation cancelled.')
```

Returns *None* on *KeyboardInterrupt* event.

argh.interaction.**safe_input**(*prompt*)
Prompts user for input. Correctly handles prompt message encoding.

11.2.5 Shell completion

Command and argument completion is a great way to reduce the number of keystrokes and improve user experience.

To display suggestions when you press `tab`, a shell must obtain choices from your program. It calls the program in a specific environment and expects it to return a list of relevant choices.

Argparse does not support completion out of the box. However, there are 3rd-party apps that do the job, such as argcomplete and python-selfcompletion.

Argh supports only argcomplete which doesn't require subclassing the parser and monkey-patches it instead. Combining *Argh* with python-selfcompletion isn't much harder though: simply use *SelfCompletingArgumentParser* instead of vanilla *ArgumentParser*.

See installation details and gotchas in the documentation of the 3rd-party app you've chosen for the completion backend.

Argh automatically enables completion if argcomplete is available (see *COMPLETION_ENABLED*). If completion is undesirable in given app by design, it can be turned off by setting `completion=False` in *argh.dispatching.dispatch()*.

Note that you don't *have* to add completion via *Argh*; it doesn't matter whether you let it do it for you or use the underlying API.

Argument-level completion

Argcomplete supports custom "completers". The documentation suggests adding the completer as an attribute of the argument parser action:

```
parser.add_argument("--env-var1").completer = EnvironCompleter
```

However, this doesn't fit the normal *Argh*-assisted workflow. It is recommended to use the *arg()* decorator:

```
@arg('--env-var1', completer=EnvironCompleter)
def func(...):
    ...
```

argh.completion.**autocomplete**(*parser*)
Adds support for shell completion via argcomplete by patching given *argparse.ArgumentParser* (sub)class.

If completion is not enabled, logs a debug-level message.

argh.completion.**COMPLETION_ENABLED = False**
Dynamically set to *True* on load if argcomplete was successfully imported.

11.2.6 Helpers

class argh.helpers.**ArghParser**(**args*, ***kwargs*)
A subclass of `ArgumentParser` with support for and a couple of convenience methods.

All methods are but wrappers for stand-alone functions *add_commands()*, *autocomplete()* and *dispatch()*.

Uses PARSER_FORMATTER.

add_commands(**args*, ***kwargs*)
Wrapper for *add_commands()*.

autocomplete()
Wrapper for *autocomplete()*.

dispatch(**args*, ***kwargs*)
Wrapper for *dispatch()*.

parse_args(*args=None*, *namespace=None*)
Wrapper for `argparse.ArgumentParser.parse_args()`. If *namespace* is not defined, `argh.dispatching.ArghNamespace` is used. This is required for functions to be properly used as commands.

set_default_command(**args*, ***kwargs*)
Wrapper for *set_default_command()*.

11.2.7 Exceptions

exception `argh.exceptions.`**AssemblingError**
Raised if the parser could not be configured due to malformed or conflicting command declarations.

exception `argh.exceptions.`**CommandError**
Intended to be raised from within a command. The dispatcher wraps this exception by default and prints its message without traceback.

Useful for print-and-exit tasks when you expect a failure and don't want to startle the ordinary user by the cryptic output.

Consider the following example:

```
def foo(args):
    try:
        ...
    except KeyError as e:
        print(u'Could not fetch item: {0}'.format(e))
        return
```

It is exactly the same as:

```
def bar(args):
    try:
        ...
    except KeyError as e:
        raise CommandError(u'Could not fetch item: {0}'.format(e))
```

This exception can be safely used in both print-style and yield-style commands (see Tutorial).

exception `argh.exceptions.`**DispatchingError**
Raised if the dispatching could not be completed due to misconfiguration which could not be determined on an earlier stage.

11.2.8 Output Processing

argh.io.**dump**(*raw_data*, *output_file*)
Writes given line to given output file. See *encode_output()* for details.

argh.io.**encode_output**(*value*, *output_file*)
Encodes given value so it can be written to given file object.

Value may be Unicode, binary string or any other data type.

The exact behaviour depends on the Python version:

Python 3.x

sys.stdout is a *_io.TextIOWrapper* instance that accepts *str* (unicode) and breaks on *bytes*.

It is OK to simply assume that everything is Unicode unless special handling is introduced in the client code.

Thus, no additional processing is performed.

Python 2.x

sys.stdout is a file-like object that accepts *str* (bytes) and breaks when *unicode* is passed to *sys.stdout.write()*.

We can expect both Unicode and bytes. They need to be encoded so as to match the file object encoding.

The output is binary if the object doesn't explicitly require Unicode.

argh.io.**safe_input**(*prompt*)
Prompts user for input. Correctly handles prompt message encoding.

11.2.9 Utilities

argh.utils.**get_arg_spec**(*function*)
Returns argument specification for given function. Omits special arguments of instance methods (*self*) and static methods (usually *cls* or something like this).

argh.utils.**get_subparsers**(*parser*, *create=False*)
Returns the `argparse._SubParsersAction` instance for given `ArgumentParser` instance as would have been returned by `ArgumentParser.add_subparsers()`. The problem with the latter is that it only works once and raises an exception on the second attempt, and the public API seems to lack a method to get *existing* subparsers.

Parameters **create** – If *True*, creates the subparser if it does not exist. Default if *False*.

11.3 Cookbook

11.3.1 Multiple values per argument

Use *nargs* from argparse by amending the function signature with the `arg()` decorator:

```
@argh.arg('-p', '--patterns', nargs='*')
def cmd(patterns=None):
    distros = ('abc', 'xyz')
    return [d for d in distros if not patterns
                                  or any(p in d for p in patterns)]
```

Resulting CLI:

```
$ app
abc
xyz

$ app --patterns
abc
xyz

$ app -p a
abc

$ app -p ab yz
abc
xyz
```

Note that you need to specify both short and long names of the argument because *@arg* turns off the "smart" mechanism.

11.4 Similar projects

Obviously, *Argh* is not the only CLI helper library in the Python world. It was created when some similar solutions already existed; more appeared later on. There are valid reasons behind maintaining most projects.

The list below is nowhere near exhausting; certain items are yet to be reviewed; the comments should have been more structured. However, it gives a picture of the alternatives.

Ideally, we'd need a table with the following columns: supports argparse; has integrated parser; requires subclassing; supports nested commands; is bound to an unrelated piece of software; involves "magic" (i.e. undermines clarity); depends on outdated libraries; has simple API; has unobtrusive API; supports Python3. Not every "yes" in this table would count as pro.

- argdeclare requires additional classes and lacks support for nested commands.
- argparse-cli requires additional classes.
- django-boss seems to lack support for nested commands and is strictly Django-specific.
- entrypoint is lightweight but involves a lot of magic and seems to lack support for nested commands.
- opster and finaloption support nested commands but are based on the outdated *optparse* library and therefore reimplement some features available in *argparse*. They also introduce decorators that don't just decorate functions but change their behaviour, which is bad practice.
- simpleopt has an odd API and is rather a simple replacement for standard libraries than an extension.
- opterator is based on the outdated *optparse* and does not support nested commands.
- clap ships with its own parser and therefore is incompatible with *clap*-agnostic code.
- plac is a very powerful alternative to *argparse*. I'm not sure if it's worth migrating but it is surely very flexible and easy to use.
- baker
- plumbum
- docopt
- aaargh

- cliff
- cement

11.5 Real-life usage

Below are some examples of applications using *argh*, grouped by supported version of Python.

Python 3:

- Aurifere

Python 2 and 3:

- Watchdog

Python 2:

- Tool
- Poni
- Pyg
- Barman
- Timetra

...and more. Well, there's probably no need to keep a complete and up-to-date list. Still, please let me know anyway if you use *argh* in your project. I'll be glad to know. :-)

11.6 Subparsers

The statement `parser.add_commands([bar, quux])` builds two subparsers named *bar* and *quux*. A "subparser" is an argument parser bound to a namespace. In other words, it works with everything after a certain positional argument. *Argh* implements commands by creating a subparser for every function.

Again, here's how we create two subparsers for commands `foo` and `bar`:

```
parser = ArghParser()
parser.add_commands([bar, quux])
parser.dispatch()
```

The equivalent code without *Argh* would be:

```
import argparse

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers()

foo_parser = subparsers.add_parser('foo')
foo_parser.set_defaults(function=foo)

bar_parser = subparsers.add_parser('bar')
bar_parser.set_defaults(function=bar)

args = parser.parse_args()
print args.function(args)
```

Now consider this expression:

```
parser = ArghParser()
parser.add_commands([bar, quux], namespace='foo')
parser.dispatch()
```

It produces a command hierarchy for the command-line expressions `foo bar` and `foo quux`. This involves "sub-subparsers". Without *Argh* you would need to write something like this (generic argparse API):

```
import sys
import argparse

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers()

foo_parser = subparsers.add_parser('foo')
foo_subparsers = foo_parser.add_subparsers()

foo_bar_parser = foo_subparsers.add_parser('bar')
foo_bar_parser.set_defaults(function=bar)

foo_quux_parser = foo_subparsers.add_parser('quux')
foo_quux_parser.set_defaults(function=quux)

args = parser.parse_args()
print args.function(args)
```

Note: You don't have to use *ArghParser*; the standard `argparse.ArgumentParser` will do. You will just need to call stand-alone functions `add_commands()` and `dispatch()` instead of *ArghParser* methods.

11.7 Contributors

Here is an inevitably incomplete list of contributors, i.e. people who have suggested features, reported bugs, submitted patches, wrote packaging scripts and generally made *Argh* better:

Andrey Mikhaylenko Author, Maintainer

Gora Khargosh Bug reports

Mika Eloranta Patches

Fabien Devaux ArchLinux package

Hannu Valtonen Debian package

Georges Dubus Python3 support fixes

Roman Ovchinnikov Debian package

thethomasw Python2.6 bug reports

Tuk Bredsdorff List of similar projects

Mike Gilbert Gentoo package; patch

Marco Nenciarini Patch for shell completion and more

Matt Black Patch re TTY

Tony Narlock Adaptation of README to GitHub

Oskari Saarenmaa Compatibility improvements

Denis Lisov Support for keyword-only arguments (Python 3)

Jörg Doppler Defaults in argument help message, raw docstrings

Paul Jacobson Defaults in argument help message, raw docstrings

Chuck Blake Support for Cython

invl Idea and basic implementation of EntryPoint

illumin-us-r3v0lution Questions and examples of setuptools integration

Joseph McCullough Patch for dev environ

Jason Dusek Patch for EntryPoint

Felix Yan Fix missing test dependencies

David Warde-Farley Bugfix

Jakub Wilk Fix spelling in docs

Brian Lee Support for signatures of funcs behind @wraps deco

...you? :-) Patches, ideas and any feedback is highly appreciated.

11.7.1 Acknowledgements

Early versions were somewhat inspired by Alexander Solovyov's opster.

Thanks to the authors of argparse for the excellent library for which Argh is merely a wrapper.

Thanks to Andrey Kislyuk for writing argcomplete and thus allowing Argh to remain compact.

Thanks to the authors of py.test, tox, virtualenv, mock and related projects (or ideas) for automating the routine and letting the developer focus on the task and enjoy TDD.

Thanks to Bitbucket team for the not-too-commercial approach to the excellent tools they provide.

11.8 Changelog

11.8.1 Version 0.26.3-dev

Backward incompatible changes:

- Dropped support for Python 2.6.

Enhancements:

- Added support for Python 3.5.
- Support introspection of function signature behind the *@wraps* decorator (issue #111).

Fixed bugs:

- When command function signature contained `**kwargs` *and* positionals without defaults and with underscores in their names, a weird behaviour could be observed (issue #104).

Other changes:

- Include the license files in manifest (PR #112).

11.8.2 Version 0.26.2

- Removed official support for Python 3.4, added for 3.5.
- Various tox-related improvements for development.
- Improved documentation.

11.8.3 Version 0.26.1

Fixed bugs:

- The undocumented (and untested) argument *dispatch(..., pre_call=x)* was broken; fixing because at least one important app depends on it (issue #63).

11.8.4 Version 0.26

This release is intended to be the last one before 1.0. Therefore a major cleanup was done. This **breaks backward compatibility**. If your code is really outdated, please read this list carefully and grep your code.

- Removed decorator *@alias* (deprecated since v.0.19).
- Removed decorator *@plain_signature* (deprecated since v.0.20).
- Dropped support for old-style functions that implicitly expected namespace objects (deprecated since v.0.21). The *@expects_obj* decorator is now mandatory for such functions.
- Removed decorator *@command* (deprecated since v.0.21).
- The *@wrap_errors* decorator now strictly requires that the error classes are given as a list (old behaviour was deprecated since v.0.22).
- The *allow_warnings* argument is removed from *argh.completion.autocomplete()*. Debug-level logging is used instead. (The warnings were deprecated since v.0.25).

Some more stuff has been scheduled **to be purged before 1.0**:

- Deprecated arguments *title*, *help* and *description* in *add_commands()* helper function. See documentation and issue #60.

Other changes:

- Improved representation of default values in the help.
- Dispatcher can be configured to skip unknown arguments (issue #57).
- Added *add_subcommands()* helper function (a convenience wrapper for *add_commands()*).
- *EntryPoint* now stores kwargs for the parser.
- Added support for default command *with* nested commands (issue #78).

  This only works with Python 3.4+ due to incorrect behaviour or earlier versions of Argparse (including the stand-alone one as of 1.2.1).

  Due to argparse peculiarities the function assignment technique relies on a special *ArghNamespace* object. It is used by default in *ArghParser* and the shortcuts, but if you call the vanilla *ArgumentParser.parse_args()* method, you now *have* to supply the proper namespace object.

Fixed bugs:

- Help formatter was broken for arguments with empty strings as default values (issue #76).

11.8.5 Version 0.25

- Added EntryPoint class as another way to assemble functions (issue #59).
- Added support for Python 3.4; dropped support for Python 3.3 (this doesn't mean that Argh is necessarily broken under 3.3, it's just that I'm not testing against it anymore).
- Shell completion warnings are now deprecated in favour of *logging*.
- The command help now displays default values of all arguments (issue #64).
- Function docstrings are now displayed verbatim in the help (issue #64).
- Argh's dispatching now should work properly in Cython.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

a

A

C

D

E

G

N

P

S

W