begins Documentation

Release 0.9

Aaron Iles

August 01, 2014

Contents

1 Features **3**

2 Table of Contents **5**
- 2.1 A short tutorial on begins . . . 5
- 2.2 Guide to using begins . . . 7
- 2.3 The begins API . . . 16

3 Issues **19**

Python Module Index **21**

Begins makes creating command line applications in Python easy.

> "The beginning is the most important part of the work."
>
> —Plato

Its decorator based API uses your function's call signatures to generate command line parsers. Subcommands, type conversion, environment variables and configuration files are all supported. Extensions encapsulate common patterns for command line configuration like logging and tracebacks.

The following short example demonstrates many of *begins* features for both Python2 and Python3.

```
import begin
import logging

@begin.start(auto_convert=True)
@begin.logging
def run(host='127.0.0.1', port=8080, noop=False):
    if noop:
        return
    logging.info("%s:%d", host, port)
```

CHAPTER 1

Features

- Small API based on function decorators.
- Generates command line options from function signatures.
- Decorate multiple functions to create sub-commands.
- Automatically convert command line arguments to expected types.
- Use environment variables and command line files to set defaults.
- Pre-packaged to extensions to simplify common tasks.

CHAPTER 2

Table of Contents

2.1 A short tutorial on begins

The purpose of *begins* is to help you to easily create command line applications without distracting you from the purpose of you application. Let's create an application together to demonstrate some of the features of *begins*. Our application will generate a series of famous Monty Python quotes.

First we will create a minimal structure for our application.:

```
import begin
@begin.start
def run():
    "Monty Python quotes for all"
```

This application will run. It doesn't need any `__name__ == "__main__"` magic. It even has help output when it's run with the `-h` flag.

Now lets make it do something more interesting.:

```
import begin
@begin.start
def run(name='Arthur', quest='Holy Grail', colour='blue', *knights):
    "Monty Python quotes for all"
```

Now our application accepts three command line flags and zero or more positional arguments. The command line flags all have default values that can be over ridden.

Currently all these options relate to different Monty Python quotes. Let us separate these into separate sub-commands.:

```
import begin
@begin.subcommand
def name(answer):
    "What is your name?"

@begin.subcommand
def quest(answer):
    "What is your quest?"

@begin.subcommand
def colour(answer):
    "What is your favourite colour?"

@begin.start
```

```
def run():
    pass
```

Now our application has three sub-commands that require exactly one positional argument. However, you may have noticed there is currently no output. We will change that now. One way to fix this is to use `print` statements `print()` functions. We are not going to do that. Instead, we are going to configure logging.:

```
import begin
import logging

@begin.subcommand
def name(answer):
    "What is your name?"
    logging.info(answer)

@begin.subcommand
def quest(answer):
    "What is your quest?"
    logging.info(answer)

@begin.subcommand
def colour(answer):
    "What is your favourite colour?"
    logging.info(answer)

@begin.start
@begin.logging
def run():
    pass
```

Now our application does something useful. But, it can only answer one question at a time. We can ask *begins* to accept multiple subcommands using a separator. We will ignore the sub-command definitions for brevity.:

```
# Sub-commands not shown for brevity.

@begin.start(cmd_delim='--')
@begin.logging
def run():
    pass
```

Lastly, let us create a new sub-command that takes a number instead of a string.:

```
@begin.subcommand
@begin.convert(speed=int)
def swallows(speed):
    "What is the wing speed of an unladden swallow?"
    logging.info("%d mph", speed)
```

You may have noticed the introduction of the `begin.convert` decorator. If the sub-command was not decorated in this way the `speed` argument would be a string. This would cause the sub-command to fail. The `begin.convert` decorator ensures that the `speed` argument will be converted to an integer first.

This is the end for this little tutorial. To learn more about *begins* you can continue reading with the user *Guide to using begins*.

2.2 Guide to using begins

2.2.1 Overview

Command line programs for *lazy* humans.

- Decorate a function to be your programs starting point.
- Generate command line parser based on function signature.
- Search system environment for option default values.

2.2.2 Why begins?

I write a lot of small programs in Python. These programs often accept a small number of simple command line arguments. Having to write command line parsing code in each of these small programs both breaks my train of thought and greatly increases the volume of code I am writting.

Begins was implemented to remove the boilerplate code from these Python programs. It's not intended to replace the rich command line processing needed for larger applications.

2.2.3 Requirements

For Python versions earlier than Python 3.3, the funcsigs package from the Python Package Index is required.

For Python version 2.6, the argparse package from the Python Package Index is also required.

Both of these dependencies are listed in the package configuration. If using Pip to install *begins* then the required dependencies will be automatically installed.

2.2.4 Installation

begins is available for download from the Python Package Index. To install using Pip

```
$ pip install begins
```

Alternatively, the latest development version can be installed directly from Github.

```
$ pip install git+https://github.com/aliles/begins.git
```

Please note that *begins* is still in an alpha state and therefore the API or behaviour could change.

2.2.5 Setting a programs starting point

The `begin.start()` function can be used as a function call or a decorator. If called as a function it returns `True` when called from the `__main__` module. To do this it inspects the stack frame of the caller, checking the `__name__` global.

This allows the following Python pattern:

```
>>> if __name__ == '__main__':
...     pass
```

To be replace with:

```
>>> import begin
>>> if begin.start():
...     pass
```

If used as a decorator to annotate a function the function will be called if defined in the `__main__` module as determined by inspecting the current stack frame. Any definitions that follow the decorated function wont be created until after the function call is complete.

Usage of `begin.start()` as a decorator looks like:

```
>>> import begin
>>> @begin.start
... def run():
...     pass
```

By deferring the execution of the function until after the remainder of the module has loaded ensures the main function doesn't fail if depending on something defined in later code.

2.2.6 Parsing command line options

If `begin.start()` decorates a function accepts parameters `begin.start()` will process the command for options to pass as those parameters:

```
>>> import begin
>>> @begin.start
... def run(name='Arther', quest='Holy Grail', colour='blue', *knights):
...     "tis but a scratch!"
```

The decorated function above will generate the following command line help:

```
usage: example.py [-h] [-n NAME] [-q QUEST] [-c COLOUR]
                  [knights [knights ...]]

tis but a scratch!

positional arguments:
  knights

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  (default: Arther)
  -q QUEST, --quest QUEST
                        (default: Holy Grail)
  -c COLOUR, --colour COLOUR
                        (default: blue)
```

In Python3, any function annotations for a parameter become the command line option help. For example:

```
>>> import begin
>>> @begin.start
... def run(name: 'What, is your name?',
...         quest: 'What, is your quest?',
...         colour: 'What, is your favourite colour?'):
...     pass
```

Will generate command help like:

```
usage: holygrail_py3.py [-h] -n NAME -q QUEST -c COLOUR

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  What, is your name?
  -q QUEST, --quest QUEST
                        What, is your quest?
  -c COLOUR, --colour COLOUR
                        What, is your favourite colour?
```

Command line parsing supports:

- positional arguments
- keyword arguments
- default values
- variable length arguments
- annotations

Command line parsing does not support variable length keyword arguments, commonly written as `**kwargs`. If variable length keyword arguments are used by the decorated function an exception will be raised.

If a parameter does not have a default, failing to pass a value on the command line will cause running the program to print an error and exit.

For programs that have a large number of options it may be preferable to only use long options. To suppress short options, pass `False` as the `short_args` keyword argument to the `begin.start` decorator:

```
>>> import begin
>>> @begin.start(short_args=False)
... def run(name='Arther', quest='Holy Grail', colour='blue', *knights):
...     "tis but a scratch!"
```

This program will not accept `-n`, `-q` or `-c` as option names.

Similarity, a large number of command line options may be better displayed in alphabetical order. This can be achieved by passing `lexical_order` as `True`:

```
>>> import begin
>>> @begin.start(lexical_order=True)
... def main(charlie=3, alpha=1, beta=2):
...     pass
```

This program will list the command line options as `alpha`, `beta`, `charlie` instead of the order in which the function accepts them.

2.2.7 Boolean options

If a command line option has a default value that is a `bool` object. (`True` or `False`) The command line option will be flags rather than an option that accepts a value. Two flags are generated, one to set a `True` value and one to set a `False` value. The two commands will be of the form `--flag` and `--no-flag`. For example:

```
>>> import begin
>>> @begin.start
... def main(enable=False, disable=True):
...     pass
```

Using `--enable` or `--no-disable` when invoking this program will invert the associated option. The options `--no-enable` and `--disable` have not effect.

2.2.8 Sub-Commands

begins supports using functions as sub-commands with the `begin.subcommand()` decorator:

```
>>> import begin
>>> @begin.subcommand
... def name(answer):
...     "What is your name?"
...
>>> @begin.subcommand
... def quest(answer):
...     "What is your quest?"
...
>>> @begin.subcommand
... def colour(answer):
...     "What is your favourite colour?"
...
>>> @begin.start
... def main():
...     pass
```

This example registers three sub-commands for the program:

```
usage: subcommands.py [-h] {colour,name,quest} ...

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

Available subcommands:
  {colour,name,quest}
    colour             What is your favourite colour?
    name               What is your name?
    quest              What is your quest?
```

The main function will always be called with the provided command line arguments. If a sub-command was chosen the associated function will also be called.

It is possible to create a sub-command with a different name from the decorated function's name. To do this pass the desired sub-command name using the `name` keyword argument:

```
>>> import begin
>>> @begin.subcommand(name='colour')
... def question(answer):
...     "What is your favourite colour?"
```

Sub-commands can also be registered with a specific named group by passing a `group` argument to the `begin.subcommand` decorator. The `begin.start()` decorator can use sub-commands from a named group by passing it a `sub_group` argument.

Similarly, sub-commands can be load from entry points by passing the name of the entry point through the `plugins` argument to the `begin.start()` decorator:

```
>>> import begin
>>> @begin.start(plugins='begins.plugin.demo')
... def main():
...     pass
```

Any functions from installed packages that are registered with the `begins.plugin.demo` entry point will be loaded as sub-commands.

2.2.9 Multiple Sub-Commands

Some commands may benefit from being able to be called with multiple subcommands on the command line. The enable multiple sub-commands a command separator value needs to be passed to be passed to `begin.start()` as the `cmd_delim` parameter:

```
>>> import begin
>>> @begin.subcommand
... def subcmd():
...     pass
...
>>> @begin.start(cmd_delim='--')
... def main():
...     pass
```

When this program is called from the command line multiple instances of the sub-command may be called if separated by the command delimiter `--`.

2.2.10 Sub-Command Context

There are use cases where it is desirable to pass state from the main function to a subsequent sub-command. To support this Begins provides the `begin.context` object. This object will have the following properties:

- `last_return`, value returned by previous command function.
- `return_values`, iterable of all return values from previous commands.
- `opts_previous`, iterable of options object used by previous commands.
- `opts_current`, options object for current command.
- `opts_next`, iterable of options object for following commands.
- **(deprecated)** `return_value`, replaced by `last_return`.

Any other properties set on the `begin.context` object will not be altered by begins.

The `last_return` property and `return_values` will always be populated, even in the value returned from the main function or a sub-command function is the `None` object. The length and order of the `return_values` will match those of `opts_previous`.

2.2.11 Environment Variables

Environment variables can be used to override the default values for command line options. To use environment variables pass a prefix string to the `begin.start()` decorator through the `env_prefix` parameter:

```
>>> import begin
>>> @begin.start(env_prefix='MP_')
... def run(name='Arther', quest='Holy Grail', colour='blue', *knights):
...     "tis but a scratch!"
```

In the example above, if an environment variable `MP_NAME` existed, it's value would be used as the default for the `name` option. The options value can still be set by explicitly passing a new value as a command line option.

2.2.12 Configuration files

Configuration files can also be used to override the default values of command line options. To use configuration files pass a base file name to the `begin.start()` decorator through the `config_file` parameter:

```
>>> import begin
>>> @begin.start(config_file='.camelot.cfg')
... def run(name='Arther', quest='Holy Grail', colour='blue', *knights):
...     "tis but a scratch!"
```

This example will look for configuration files named `.camelot.cfg` in the current directory and/or the user's home directory. A command line option's default value can be changed by an option value in a configuration file. The configuration section used matches the decorated function's name by default. This can be changed by passing a `config_section` parameter to `begin.start()`:

```
>>> import begin
>>> @begin.start(config_file='.camelot.cfg', config_section='camelot')
... def run(name='Arther', quest='Holy Grail', colour='blue', *knights):
...     "tis but a scratch!"
```

In this second example the section `camelot` will be used instead of a section named `run`.

2.2.13 Argument type casting

Command line arguments are always passed as strings. Sometimes thought it is more convenient to receive arguments of different types. For example, this is a possible function for starting a web application:

```
>>> import begin
>>> @begin.start
... def main(host='127.0.0.1', port='8080', debug='False'):
...     port = int(port)
...     debug = begin.utils.tobool(debug)
...     "Run web application"
```

Having to convert the `port` argument to an integer and the `debug` argument to a boolean is additional boilerplate code. To avoid this *begins* provides the `begin.convert()` decorator. This decorator accepts functions as keyword arguments where the argument name matches that of the decorator function. These functions are used to convert the types of arguments.

Rewriting the example above using the `begin.convert()` decorator:

```
>>> import begin
>>> @begin.start
... @begin.convert(port=int, debug=begin.utils.tobool)
... def main(host='127.0.0.1', port=8080, debug=False):
...     "Run web application"
```

The module `begin.utils` contains useful functions for converting argument types.

2.2.14 Automatic casting

For simple, built-in types *begins* can automatically type cast arguments. This is achieved by passing the parameter `_automatic` to `begin.convert()`:

```
>>> import begin
>>> @begin.start
... @begin.convert(_automatic=True)
```

```
... def main(host='127.0.0.1', port=8080, debug=False):
...     "Run web application"
```

This example is functionally equivalent to the example above.

Automatic type casting works for the following built-in types.

- `int` or `long`
- `float`
- `boolean`
- `tuple` or `list`

Additional casting functions can be provided with the same call to the `begin.convert()` decorator.

Alternatively, use of `begin.convert()` can be dispensed by passing `True` to `begin.start()` via the `auto_convert` parameter:

```
>>> import begin
>>> @begin.start(auto_convert=True)
... def main(host='127.0.0.1', port=8080, debug=False):
...     "Run web application"
```

Again, this example is functionally equivalent to the example above.

The limitation of using `auto_convert` is that it is not longer possible to provide additional casting functions.

2.2.15 Command Line Extensions

There are behaviours that are common to many command line applications, such as configuring the `logging` and `cgitb` modules. *begins* provides function decorators that extend a program's command line arguments to configure these modules.

- `begin.tracebacks()`
- `begin.logging()`

To use these decorators they need to decorate the main function before `begin.start()` is applied.

Tracebacks

The `begin.tracebacks()` decorator adds command line options for extended traceback reports to be generated for unhandled exceptions:

```
>>> import begin
>>> @begin.start
... @begin.tracebacks
... def main(*message):
...     pass
```

The example above will now have the following additional argument group:

```
tracebacks:
  Extended traceback reports on failure

  --tracebacks   Enable extended traceback reports
  --tbdir TBDIR  Write tracebacks to directory
```

Passing `--tracebacks` will cause extended traceback reports to be generated for unhandled exceptions.

Traceback options may also be set using configuration files, if Configuration files are supported. The follow options are used.

- `enabled`: use any of `true`, `t`, `yes`, `y`, `on` or `1` to enable tracebacks.
- `directory`: write tracebacks to this directory.

Options are expected to be in a `tracebacks` section.

Logging

The `begin.logging()` decorator adds command line options for configuring the logging module:

```
>>> import logging
>>> import begin
>>> @begin.start
... @begin.logging
... def main(*message):
...     for msg in message:
...         logging.info(msg)
```

The example above will now have two additional optional arguments as well as an additional argument group:

```
optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Increse logging output
  -q, --quiet           Decrease logging output

logging:
  Detailed control of logging output

  --loglvl {DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        Set explicit log level
  --logfile LOGFILE     Ouput log messages to file
  --logfmt LOGFMT       Log message format
```

The logging level defaults to `INFO`. It can be adjusted by passing `--quiet`, `--verbose` or explicitly using `--loglvl`.

The default log format depends on whether log output is being directed to standard out or file. The raw log text is written to standard out. The log message written to file output includes:

- Time
- Log level
- Filename and line number
- Message

The message format can be overridden using the `--logfmt` option.

Logging options may also be set using configuration files, if Configuration files are supported. The follow options are used.

- `level`: log level, must be one of `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`.
- `file`: output log messages to this file.
- `format`: log message format.

Options are expected to be in a `logging` section.

2.2.16 Command Line Formatting

The default argparse help formatter may not always meet your needs. An alternate formatter can be provided using the `formatter_class` argument to `begin.start()`:

```
>>> import begin, argparse
>>> @begin.start(formatter_class=argparse.RawTextHelpFormatter)
... def main():
...     pass
```

Any of the formatter classes provided by the argparse module can be used.

Alternatively, `begin.formatters` provides a mechanism to compose new formatter class according to your requirements.:

```
>>> from begin import formatters
>>> formatter_class = formatters.compose(formatters.RawDescription, formatters.RawArguments)
```

The following mixin classes are provided for use with `begin.formatters.compose()`

- RawDescription
- RawArguments
- ArgumentDefaults
- RemoveSubcommandsLine

One or more of these may be passed to `begin.formatters.compose()` to create a new formatter class.

2.2.17 Entry Points

The setuptools package supports automatic script creation to automatically create command line scripts. These command line scripts use the entry points system from setuptools.

To support the use of entry points, functions decorated by `begin.start()` have an instance method called `start()` that must be used to configure the entry point:

```
setup(
    # ...
    entry_points = {
        'console_scripts': [
            'program = package.module:main.start'
        ]
    }
```

Use of the `start()` method is required because the main function is not called from the `__main__` module by the entryp points system.

2.2.18 Issues

Any bug reports or feature requests can be made using Github's issues system.

2.3 The begins API

Convenience function for starting Python programs

begin.**start**(*func=None*, ***kwargs*)

Return True if called in a module that is executed.

Inspects the '__name__' in the stack frame of the caller, comparing it to '__main__'. Thus allowing the Python idiom:

```
>>> if __name__ == '__main__':
...     pass
```

To be replace with:

```
>>> import begin
>>> if begin.start():
...    pass
```

Can also be used as a decorator to register a function to run after the module has been loaded.

```
>>> @begin.start
... def main():
...     pass
```

This also inspects the stack frame of the caller, choosing whether to call function immediately. Any definitions following the function wont be called until after the main function is complete.

If used as a decorator, and the decorated function accepts either positional or keyword arguments, a command line parser will be generated and used to parse command line options.

```
>>> @begin.start
... def main(first, second=''):
...     pass
```

This will cause the command line parser to accept two options, the second of which defaults to ''.

Default values for command line options can also be set from the current environment, using the uppercased version of an options name. In the example above, the environment variable 'FIRST' will set a default value for the first argument.

To use a prefix with expected environment variables (for example, to prevent collisions) give an 'env_prefix' argument to the decorator.

```
>>> @begin.start(env_prefix='PY_')
... def main(first, second=''):
...     pass
```

The environment variable 'PY_FIRST' will be used instead of 'FIRST'.

2.3.1 Utilities

Utility functions for begins

begin.utils.**tobool**(*value*)

Convert a string representation of truth to True or False.

True values are 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. Raises ValueError if 'value' is anything else.

begin.utils.**tolist**(*value=None*, *sep=','*, *empty_strings=False*)
Convert a string to a list.

The input string is split on the separator character. The default separator is ','. An alternative separator may be passed as the 'sep' keyword. If no string value is provided a function is returned that splits a string on the provided separator, or default if none was provided. Any empty strings are removed from the resulting list. This behaviour can be changed by passing True as the 'empty_strings' keyword argument.

2.3.2 Help Formatters

Help formatters for use with argparse

class begin.formatters.**ArgumentDefaults**
Help message formatter which adds default values to argument help.

Based on argparse.ArgumentDefaultsHelpFormatter from Python standard library Copyright 2001-2014 Python Software Foundation; All Rights Reserved

class begin.formatters.**RawArguments**
Help message formatter which retains formatting of all argument text.

Based on argparse.RawTextHelpFormatter from Python standard library Copyright 2001-2014 Python Software Foundation; All Rights Reserved

class begin.formatters.**RawDescription**
Help message formatter which retains any formatting in descriptions.

Based on argparse.RawDescriptionHelpFormatter from Python standard library Copyright 2001-2014 Python Software Foundation; All Rights Reserved

class begin.formatters.**RemoveSubcommandsLine**
Removes line of subcommand names from help output.

Based on Jeppe Ledet-Pederson's solution for hiding metavar in command listing. http://stackoverflow.com/a/13429281

begin.formatters.**compose**(**mixins*, ***kwargs*)
Compose a new help formatter class for argparse.

Accepts a variable number of mixin class and uses these as base classes with multiple inheritance against argparse.HelpFormatter to create a new help formatter class.

CHAPTER 3

Issues

If you encounter problems, please refer to *Issues* from the guide.

b

begin, 16
begin.formatters, 17
begin.utils, 16