dryparse

Release 0.0.0

Haris Gušić

Jul 02, 2022

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Installation

```
$ pip install dryparse
```

1.2 Creating a simple command

CHAPTER

TWO

OBJECT MODEL

The decorators are useful for commands that are simple and follow basic CLI conventions. Sometimes you want to extend your CLI with custom features. This is where the object model comes in.

Each concept in a commandline program is represented by an object in the *dryparse.objects* module. Thus, each command is represented by an instance of *Command*, each option by an *Option*, etc.

2.1 The fundamentals

The simplest way to create a command is:

```
git = Command("git", desc="A version control software")
```

Adding options is super easy:

```
git.paginate = Option("-p", "--paginate")
```

Note: The `git.paginate` attribute didn't exist before. We created it dynamically by assigning a value to it.

By default, options have a type of `bool`. This means that when the option is specified on the command line it will have a value of `True`, and `False` otherwise.

Let's create an option of type `int`:

```
git.retries = Option("-r", "--retries", type=int)
```

This option will expect an argument to be specified via the command line. The argument is automatically converted to the type we specified (in this case `int`).

Note: All options except for `bool` and some *special types* take CLI arguments, and those arguments are automatically converted to the specified type.

Note that commands include a `--help` option by default, via a `help` attribute that is just like any other attribute. You can delete it if you don't need it:

```
del git.help
```

2.2 Adding a subcommand

Adding a subcommand is just as easy as adding an option:

```
git.commit = Command("commit", desc="Record changes to the repository")
git.checkout = Command("checkout", desc="Switch branches or restore working tree files")
```

2.3 Defining positional arguments

```
git.add.args = Arguments([])
```

These use cases are simple, but *Arguments* has so much more to offer. Take a look at its API documentation.

2.4 Root command

CLI programs usually include a `--version` option in their root command. While you can add this option yourself, we provide *RootCommand* as a convenience:

```
git = RootCommand("git", version="0.1.0", desc="A version control software")
```

CHAPTER

THREE

CUSTOMIZING HELP

Dryparse generates standard help out of the box, but it also provides a hierarchical representation of a help message via *Help*. You can obtain a help object for any of: *Command*, *Option*, *Group*.

Todo: Group help is not implemented yet.

```
git_help = Meta(git).help
dryparse.parse()
```

For example, the default help message for the `git` subcommand we've been building so far would be:

```
A version control software

Usage: git [-h] []
```

You can easily convert this to a help message string using:

```
str(git_help)
# or
git_help.text
```

CHAPTER

FOUR

API

4.1 dryparse.help

Help module for dryparse objects.

class Help

Hierarchical representation of a help message. You can customize every individual part or subpart of it, or its entirety.

text: str

The entire help text.

Overriding this property makes all the other properties obsolete.

class CommandHelp(*command:* Command)

Object that represents a command's help message organized as a hierarchy.

signature: str

Describes the command signature when listed as a subcommand in the help message of another command.

Defaults to the command name.

desc: *dryparse.help.CommandDescription*

Command description.

You can assign this to be a `str` or *CommandDescription*, but you will always get a *CommandDescription* when you try to access this as an attribute.

sections: *dryparse.help.HelpSectionList*

Sections of this help message.

Default implementation returns a standard command section list including "usage", "subcommands", etc.

section_separator: str

Separator between sections of this help message.

listing: *dryparse.help.HelpEntry*

Text that appears when this subcommand is listed as a subcommand in the help message of another command.

text: str

class OptionHelp(*option:* Option)

Attributes

override_class: Optional[Type[OptionHelp]] When instantiating an object of type *OptionHelp*, return an instance of `override_class` instead.

desc: str

Option description.

argname: str

Name of the argument to this option.

signature: str

Option signature.

Default value:

- `-o ARGNAME, --option ARGNAME`, if the option takes an argument
- `-o ARGNAME`, if the option only has a short text
- `--option ARGNAME`, if the option only has a long text
- `--o ARGNAME`, if the option only has a short text
- `ARGNAME` is omitted if the option takes no argument

hint: str

Hint for the option that appears in the "usage" section of a command.

Default value:

- `[-o ARGNAME]`, if the option has a short text
- `[--option ARGNAME]`, if the option has no short text
- `ARGNAME` is omitted if the option does not take an argument

text: str

class GroupHelp(*group: Union[str,* Group*]*)

Help for a *Group* object.

text: str

class HelpSection(*name_or_group: Union[str,* Group*]*)

Help section, with a headline and content.

name: str

Section name.

group: Optional[*dryparse.objects.Group*]

Group that this section refers to.

headline: str

Section headline.

Default value: `f"{self.name}:"`.

content: str

Section content, excluding headline.

indent: int

Indent for the content of this section.

active: bool

Controls whether the section is displayed.

Default value: `True`

text: str

Entire help text.

class HelpSectionList(*iterable=()*, */*)

A glorified list of help sections.

Behaves exactly like a normal list, with the addition that you can also access individual help sections as attributes, not just by indexing. Sections added using regular list functions can only be accessed by index (and iterators, etc.), while sections added by attribute assignment can also be accessed as attributes. In the latter case, the index is determined by the number of sections that existed before the section was added.

Examples

```
>>> sections = HelpSectionList()
>>> # Create unnamed section at index 0
>>> sections.append(HelpSection("Usage"))
>>> # Create section named usage - index automatically set to be 1
>>> sections.commands = HelpSection("Commands")
>>> print(sections[0].text)
Usage:
>>> print(sections.usage.text)
Commands:
>>> print(sections[1])
Commands:
```

class HelpEntry(*signature*, *desc*)

Represents an option or subcommand entry in a help message.

signature_width: int

Width of signature after padding.

Default: 32 characters.

padded_signature: str

Option signature padded until *signature_width*.

text: str

Entire help text for this entry.

class CommandDescription(*long*, *brief=None*)

Command description that can hold both a long and a brief version.

Attributes

long: str Long description. Used in the help text of the command at hand.

brief: str Brief description. Describes this command when it appears as a subcommand in another command's help text. Falls back to `long`.

4.2 dryparse.objects

Object model of a command line program.

4.2.1 Option

class Option(*short: str = ''*, *long: str = ''*, *argname: ~typing.Optional[str] = None*, *default=None*, *argtype: type = <class 'bool'>*, *desc: ~typing.Optional[str] = None*)

Parameters

- **short** (*str*) – Regex pattern that the short version of this option should match against. Usually this is a hyphen followed by a single letter (e.g. `-s`).
- **long** (*str*) – Regex pattern that the long version of this option should match against. Usually this is two hyphens followed by multiple letters. (e.g. `--long`)

Attributes

help: OptionHelp Customizable help object.

build(*option: Optional[str] = None*)

Each time this option is specified on the command line, this method is called. The positional arguments in the function signature determine what formats are acceptable for the option.

The build function can be assigned by `option.build =`.

Parameters **option** – The exact way the option was specified. This is useful when the option text is specified as a regex, or when you want to know if the option was specified using its short or long format.

Examples

Assume that the long version of the option is `--option`.

1. If the signature is `build(self, **kwargs)`, the option is a `bool` option and can be specified as `--option`.
2. If the signature is `build(self, arg, **kwargs)`, the option can be specified as `--option ARG`

4.2.2 Command

class Command(*name*, *regex=None*, *desc: Optional[str] = None*)

A CLI command.

You can assign arbitrary attributes dynamically. Only attributes of types *Option*, *Command*, *Group* and others from *dryparse.objects* have special meaning to the parser.

Examples

```
>>> docker = Command("docker")
>>> docker.context = Option("-c", "--context")
>>> docker.run = Command("run", desc="Run a command in a new container")
```

__call__(**args*, *help=None*, ***kwargs*)

Execute the command. Unless overridden, this will process special options like `help`, and handle subcommands.

4.2.3 RootCommand

class RootCommand(*name*, *regex=None*, *desc=''*, *version='0.0.0'*)

Command that corresponds to the program itself.

Parameters **version** (*str*) – Version of the program that is printed when the `--version` option is given.

4.2.4 Group

class Group(*name: str*)

A group of commands or options.

4.2.5 Arguments

class Arguments(**pattern: Union[type, Tuple[type, Union[int, ellipsis, range]]]*)

Specification for positional arguments of a command.

Positional arguments are specified on the command line as regular strings, but usually we want to restrict the number of allowed arguments, their data types, add custom validation, etc. An instance of *Arguments* holds a pattern of acceptable argument types. The arguments are converted (and effectively validated) using *convert()*.

Attributes

pattern Determines the number of arguments and their types. Each item of this list represents an argument or multiple arguments of a given type. The order of arguments is important.

There are two acceptable formats for each entry:

- *type*: Accepts a single argument of the given type.
- (*type*, *number*): Accepts *number* arguments of type *type*.

Note that *number* can be an `int`, `...` (ellipsis) or `range`. An `int` specifies a fixed number of required arguments. Ellipsis is a special value meaning *zero or more arguments*. A range specifies a range of acceptable argument numbers.

values The list of argument values held by this instance. This attribute is assigned when you call *assign()*, and will be `None` until then.

Notes

- `(type, ...)`- and `(type, range)`- style patterns cannot be followed by further patterns. Instead, you should implement a custom converter function.
- For boolean types, you might want to use *Bool* instead of `bool`, because of potentially undesired behaviors like `bool("false") == True`, etc.

Examples

Here's an exhaustive list of example use cases:

```
>>> # Single argument of type int
>>> Arguments(int)
>>> # Two arguments of type bool
>>> Arguments((bool, 2))
>>> # Single int and two bools
>>> Arguments(int, (bool, 2))
>>> # Zero or more strings
>>> Arguments((str, ...))
>>> # Same as the above
>>> Arguments()
>>> # One or more strings
>>> Arguments(str, (str, ...))
>>> # Between 2 and 4 strings
>>> Arguments((str, range(2, 4)))
>>> # One int and zero or more strings
>>> Arguments(int, (str, ...))
>>> # ERROR: there can be no patterns after a (type, ...)-style pattern
>>> Arguments(int, (int, ...), str)
>>> # ERROR: there can be no patterns after a (type, range)-style pattern
>>> Arguments(int, (int, range(1, 2)), str)
```

convert(*args: Sequence[str]*, *allow_extra_args=False*) → Union[List[Any], Any]

Convert (and consequently validate) a list of `args` that were specified on the command line to a list of arguments conforming to `pattern`.

If the conversion of any of the arguments throws an exception, the conversion (and validation) will fail. (TODO exception)

If the pattern only expects one argument, then the single parsed argument will be returned, instead of a list with one element.

Parameters

- **args** – Arguments to convert.
- **allow_extra_args** – Do not raise an exception if there are more `args` than can fit into `self.pattern`.

assign(*args: List[str]*, *allow_extra_args=False*)

Assign a set of arguments specified on the command line to be held by this instance.

The arguments are converted and validated using *convert()* in order to conform to `pattern`.

Parameters **allow_extra_args** – See *convert()*.

Returns

List of the converted arguments.

4.2.6 Advanced

Meta

class Meta(**args*, ***kwargs*)

Meta wrapper for *Command* that can be used to access special attributes of *Command*.

Notes

- Do not modify the `options`, `command`, `subcommands` and `argument_aliases` attributes.

call(**args*, ***kwargs*)

Callback function for when this command is invoked.

You can freely assign this method on an instance of *Meta*. Just keep in mind that the signature must support all arguments and options that can be passed to it after the command is parsed. The function can but need not have a `self` argument; but if it does, it must be the first. This argument can be used to access the associated *ResolvedCommand* object.

set_callback(*func: Callable[[*Command*], Any]*)

Set the callback function to be called when this command is invoked.

When the command is parsed, the callback will be called with all the CLI arguments passed as arguments to the callback.

ResolvedCommand

class ResolvedCommand(*command:* Command, *deepcopy=True*)

Wrapper around *Command* that provides access to option values and argument values as if they were regular attributes.

Examples

```
>>> # Initialize a simple command with an option
>>> cmd = Command("test")
>>> cmd.option = Option("--option", default="DEFAULT")
>>> # Convert Command into a ResolvedCommand
>>> parsed_cmd = ResolvedCommand(cmd)
>>> print(parsed_cmd.option)
```

(continues on next page)

(continued from previous page)

```
DEFAULT
>>> # Assignment works like with a regular command
>>> parsed_cmd.option = "NON_DEFAULT"
>>> print(parsed_cmd)
NON_DEFAULT
```

4.3 dryparse.decorators

The decorator API.

command(*func: Callable[[...], Any]*)

Take a callable and turn it into a *Command* object.

Each positional argument will be converted to an *Arguments* object whose `pattern` will be generated using the type annotation.

Each keyword argument will be converted to an *Option* object, again using the information from the type annotation. If the annotation is an instance of *Option*, this instance will be used directly.

IMPORTANT: Type annotations must be actual types and not `Union`, `Any` etc. This is required because the type will used to convert CLI arguments into their Python representations. *As a special case*, the annotation for keyword arguments can be an instance of *Option*.

Notes

- `func` will become the *call* attribute associated with the *Command* object returned by this decorator. You should read the documentation of *call*.
- If a parameter is neither annotated nor has a default value, its type is assumed to be `str`.
- The text for specifying an option on the command line is derived from the argument name in the following way:
  - The short text is formed by prepending a `-` before the first character of the argument name. Note that if multiple argument names start with the same character, only the first one will get a short text.

    Example: `recursive` becomes `-r`.
  - The long text is formed by prepending `--` before the argument name, additionally replacing all `_` characters with `-`.

    Example: `work_dir` becomes `--work-dir`.

Raises *AnnotationMustBeTypeOrSpecialError* – If a type annotation is not a `type`, or an instance of *Option*, the latter only being allowed for keyword arguments.

See also:

dryparse.objects.Meta.call

subcommand(*func: Callable[[*Command*], Any]*)

Decorator used to register a subcommand inside a class's context.

4.4 dryparse.types

Module containing special option types.

class OptionType(*value: str*)

Base class for all option value types.

You can create custom types by subclassing this class.

Attributes

takes_argument: bool Whether the option value is specified as an argument to the option or is derived from the presence of the option itself.

value: Any The option argument converted to an arbitrary type.

class Counter(*value: str*)

A counter that increments each time an option is specified.

class Bool(*value: str*)

A bool type that understands `"true"`, `"True"`, `"false"`, `"False"` and empty string.

4.5 dryparse.parser

Functions for parsing the command line, i.e. converting command line arguments into their object representations.

parse(*command:* Command, *args: Optional[List[str]] = None*)

Parse `args` into `command`.

If unspecified, `args` will fall back to `sys.argv`.

parse_arg(*command:* Command, *arg: str*) → Union[Tuple[Optional[*Option*], Optional[Any]], *Command*]

Parse `arg` using the scheme from `command`.

The argument can be an option (or option + value), a positional argument or a subcommand. For options, all the usual ways of specifying a CLI option are supported:

- Long version: `--long`/`--long=`<value> for bool/non-bool options
- Short version: `-s`/`-s`<value> for bool/non-bool options

Returns

(option, value) Option object and its value, or `None` if the value was not specified in `arg` (which means that the value must be found in the next argument on the command line, unless it's a bool option).

Notes

- This function does not take into consideration the case when the option name is specified as one argument and the option value as another argument.

4.6 dryparse.context

Context information.

class Context(*args=None*, *command_arg_index=None*)

Context information, mainly from the parser.

property args: Optional[List[str]]

Command line arguments of the current context.

property command_arg_index: int

Index in *args* of the currently parsed command.

context = <dryparse.context.Context object>

Use this to query context information.

4.7 dryparse.util

Utility functions and objects used throughout dryparse.

parse_str(*text: str*, *target_type: type*)

Parse string into primitive type `type`.

first_token_from_regex(*regex: str*) → Pattern

Get first valid token from regular expression `regex`.

The first valid token is the smallest substring taken from the beginning of `regex` that forms a valid regex by itself.

class reassignable_property(*getter: Callable[[Any], Any]*)

Property whose getter function can be assigned per instance.

Caveats

- The getter must behave as if its single parameter is the only information it knows about the instance that owns this property. This is necessary to properly facilitate deep copying. (TODO: enhance and clarify example)

  Example:

```
>>> class C:
>>>     @reassignable_property
>>>     def prop(self):
>>>         return "default_value"
>>>
>>>     def __init__(self, value):
>>>         self.value = value
```

(continues on next page)

(continued from previous page)

```
>>>         # Correct:
>>>         self.prop = lambda self_: self_.value
>>>         # Wrong:
>>>         self.prop = lambda _: self.value
```

verify_function_callable_with_args(*func*, **args*, ***kwargs*)

Verify if `func(*args, **kwargs)` is a valid call without actually calling the function. If yes, do nothing, else raise an exception.

Raises *dryparse.errors.CallbackDoesNotSupportAllArgumentsError* – If the verification fails.

4.8 dryparse.errors

All errors that can be raised directly by dryparse.

exception DryParseError

Base class for all dryparse exceptions.

exception OptionRequiresArgumentError(*option: Optional[str] = None*)

exception OptionDoesNotTakeArgumentsError(*option: Optional[str] = None*)

exception OptionArgumentTypeConversionError(*argument: Optional[str] = None*, *argtype: Optional[type] = None*)

exception InvalidArgumentPatternError

exception PatternAfterFlexiblePatternError

exception ArgumentConversionError(*reason: Optional[str] = None*, *arguments: Optional[Sequence[str]] = None*, *index: Optional[int] = None*)

exception VariadicKwargsNotAllowedError

exception ValueConversionError

exception CallbackDoesNotSupportAllArgumentsError

exception NotEnoughPositionalArgumentsError

exception TooManyPositionalArgumentsError

exception ReadOnlyAttributeError(*name: str*)

exception AnnotationMustBeTypeOrSpecialError(*param: Parameter*)

exception SelfNotFirstArgumentError

If you have read the introductory sections, then you know that there is a decorator API and an object API. The object API is the basis for everything, and the decorator API is just syntactic sugar around it.

CHAPTER

FIVE

CONTRIBUTING

5.1 Documentation

- Shell code must be added like this:

```
.. prompt:: bash

   shell command here
```

- When adding python code examples, dryparse objects must link to their corresponding documentation:

  Correct

  Wrong

  Source:

  ```
  .. autolink-preface:: from dryparse.objects import Command

  .. code:: python

     cmd = Command("test")
  ```

  Result:

  ```
  cmd = Command("test")
  ```

  Source:

  ```
  .. code:: python

     cmd = Command("test")
  ```

  Result:

  ```
  cmd = Command("test")
  ```

  For more info, see the documentation of sphinx-codeautolink.

In a nutshell, dryparse is a CLI parser that makes it easy to turn regular functions and objects into command line commands and options. It works out of the box, with default behaviors that follow established practices. In addition, it provides excellent customizability and an object model that gives you the power to do anything you want with it.

As an appetizer, let's try to recreate the ubiquitous `cp` command:

```
@dryparse.command
def cp(
   *files, link=False, force=False, target_directory: str = None
):
   """
   Copy files and directories
   """
   ... # Logic goes here
```

Do this in your program's entrypoint:

```
dryparse.parse(cp, sys.argv)
```

When someone runs this in the shell:

```
cp --link -r --target-directory "/tmp/" source/ b.txt
```

this will run in the python world:

```
cp("source/", "b.txt", link=True, recursive=True, target_directory="/tmp/")
```

This works out of the box too (help is automatically generated from the function docstring):

```
$ cp --help
```

A more holistic example:

```
$ docker run -ite ENVVAR1=1 --env ENVVAR2=2     --volume=/:/mnt:ro -v /home:/h             alpine sh
```

Hint:

`-i` is short for `--interactive`
`-t` is short for `--tty`
`-e` is short for `--env`
`-v` is short for `--volume`

```
docker.run("alpine", "sh",
           interactive=True,
           tty=True,
           env=[("ENVVAR1", "1"), ("ENVVAR2", "2")],
           volume=[("/", "/mnt", "ro"), ("/home", "/h")])
```

CHAPTER

SIX

PROBLEMS

- When a command contains a subcommand and option with the same name

---

Todo: Structure

- Walkthrough
- Topic-based guide
- Advanced use cases
- Api

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V