argparse tutorial

Release 0.7.1

Peter Melnichenko

Jul 08, 2020

Contents

Contents:

CHAPTER 1

Creating and using parsers

The `argparse` module is a function which, when called, creates an instance of the Parser class.

```
-- script.lua
local argparse = require "argparse"
local parser = argparse()
```

`parser` is now an empty parser which does not recognize any command line arguments or options.

1.1 Parsing command line arguments

`:parse([argv])` method of the Parser class returns a table with processed data from the command line or `argv` array.

```
local args = parser:parse()
```

After this is executed with `lua script.lua`, `args` is an empty table because the parser is empty and no command line arguments were supplied.

1.1.1 Error handling

If the provided command line arguments are not recognized by the parser, it will print an error message and call `os.exit(1)`.

```
$ lua script.lua foo
```

```
Usage: script.lua [-h]

Error: too many arguments
```

If halting the program is undesirable, `:pparse([args])` method should be used. It returns boolean flag indicating success of parsing and result or error message.

An error can raised manually using `:error()` method.

```
parser:error("manual argument validation failed")
```

```
Usage: script.lua [-h]

Error: manual argument validation failed
```

1.1.2 Help option

As the automatically generated usage message states, there is a help option `-h` added to any parser by default.

When a help option is used, parser will print a help message and call `os.exit(0)`.

```
$ lua script.lua -h
```

```
Usage: script.lua [-h]

Options:
   -h, --help            Show this help message and exit.
```

1.1.3 Typo autocorrection

When an option is not recognized by the parser, but there is an option with a similar name, a suggestion is automatically added to the error message.

```
$ lua script.lua --hepl
```

```
Usage: script.lua [-h]

Error: unknown option '--hepl'
Did you mean '--help'?
```

1.2 Configuring parsers

Parsers have several properties affecting their behavior. For example, `description` and `epilog` properties set the text to be displayed in the help message after the usage message and after the listings of options and arguments, respectively. Another is `name`, which overwrites the name of the program which is used in the usage message (default value is inferred from command line arguments).

There are several ways to set properties. The first is to chain setter methods of Parser object.

```
local parser = argparse()
   :name "script"
   :description "A testing script."
   :epilog "For more info, see http://example.com"
```

The second is to call a parser with a table containing some properties.

```
local parser = argparse() {
   name = "script",
   description = "A testing script.",
   epilog "For more info, see http://example.com."
}
```

Finally, `name`. `description` and `epilog` properties can be passed as arguments when calling a parser.

```
local parser = argparse("script", "A testing script.", "For more info, see http://
↪example.com.")
```

CHAPTER 2

Adding and configuring arguments

Positional arguments can be added using `:argument(name, description, default, convert, args)` method. It returns an Argument instance, which can be configured in the same way as Parsers. The `name` property is required.

This and the following examples show contents of the result table returned by *parser:parse()* when the script is executed with given command-line arguments.

```
parser:argument "input"
```

```
$ lua script.lua foo
```

```
{
   input = "foo"
}
```

The data passed to the argument is stored in the result table at index `input` because it is the argument's name. The index can be changed using `target` property.

2.1 Setting number of consumed arguments

`args` property sets how many command line arguments the argument consumes. Its value is interpreted as follows:

Value	Interpretation
Number `N`	Exactly `N` arguments
String `A-B`, where `A` and `B` are numbers	From `A` to `B` arguments
String `N+`, where `N` is a number	`N` or more arguments
String `?`	An optional argument
String `*`	Any number of arguments
String `+`	At least one argument

If more than one argument can be consumed, a table is used to store the data.

```
parser:argument("pair", "A pair of arguments.")
   :args(2)
parser:argument("optional", "An optional argument.")
   :args "?"
```

```
$ lua script.lua foo bar
```

```
{
   pair = {"foo", "bar"}
}
```

```
$ lua script.lua foo bar baz
```

```
{
   pair = {"foo", "bar"},
   optional = "baz"
}
```

2.2 Setting argument choices

The `choices` property can be used to restrict an argument to a set of choices. Its value is an array of string choices.

```
parser:argument "direction"
   :choices {"north", "south", "east", "west"}
```

```
$ lua script.lua foo
```

```
Usage: script.lua [-h] {north,south,east,west}

Error: argument 'direction' must be one of 'north', 'south', 'east', 'west'
```

CHAPTER 3

Adding and configuring options

Options can be added using `:option(name, description, default, convert, args, count)` method. It returns an Option instance, which can be configured in the same way as Parsers. The `name` property is required. An option can have several aliases, which can be set as space separated substrings in its name or by continuously setting `name` property.

```
-- These lines are equivalent:
parser:option "-f" "--from"
parser:option "-f --from"
```

```
$ lua script.lua --from there
$ lua script.lua --from=there
$ lua script.lua -f there
$ lua script.lua -fthere
```

```
{
   from = "there"
}
```

For an option, default index used to store arguments passed to it is the first "long" alias (an alias starting with two control characters, typically hyphens) or just the first alias, without control characters. Hyphens in the default index are replaced with underscores. In the following table it is assumed that `local args = parser:parse()` has been executed.

Option's aliases	Location of option's arguments
`-o`	`args.o`
`-o --output`	`args.output`
`-s --from-server`	`args.from_server`

As with arguments, the index can be explicitly set using `target` property.

3.1 Flags

Flags are almost identical to options, except that they don't take an argument by default.

```
parser:flag("-q --quiet")
```

```
$ lua script.lua -q
```

```
{
   quiet = true
}
```

3.2 Control characters

The first characters of all aliases of all options of a parser form the set of control characters, used to distinguish options from arguments. Typically the set only consists of a hyphen.

3.3 Setting number of consumed arguments

Just as arguments, options can be configured to take several command line arguments.

```
parser:option "--pair"
   :args(2)
parser:option "--optional"
   :args "?"
```

```
$ lua script.lua --pair foo bar
```

```
{
   pair = {"foo", "bar"}
}
```

```
$ lua script.lua --pair foo bar --optional
```

```
{
   pair = {"foo", "bar"},
   optional = {}
}
```

```
$ lua script.lua --optional=baz
```

```
{
   optional = {"baz"}
}
```

Note that the data passed to `optional` option is stored in an array. That is necessary to distinguish whether the option was invoked without an argument or it was not invoked at all.

3.4 Setting argument choices

The `choices` property can be used to specify a list of choices for an option argument in the same way as for arguments.

```
parser:option "--format"
   :choices {"short", "medium", "full"}
```

```
$ lua script.lua --format foo
```

```
Usage: script.lua [-h] [--format {short,medium,full}]

Error: argument for option '--format' must be one of 'short', 'medium', 'full'
```

3.5 Setting number of invocations

For options, it is possible to control how many times they can be used. argparse uses `count` property to set how many times an option can be invoked. The value of the property is interpreted in the same way `args` is.

```
parser:option("-e --exclude")
   :count "*"
```

```
$ lua script.lua -eFOO -eBAR
```

```
{
   exclude = {"FOO", "BAR"}
}
```

If an option can be used more than once and it can consume more than one argument, the data is stored as an array of invocations, each being an array of arguments.

As a special case, if an option can be used more than once and it consumes no arguments (e.g. it's a flag), than the number of invocations is stored in the associated field of the result table.

```
parser:flag("-v --verbose", "Sets verbosity level.")
   :count "0-2"
   :target "verbosity"
```

```
$ lua script.lua -vv
```

```
{
   verbosity = 2
}
```

CHAPTER 4

Mutually exclusive groups

A group of arguments and options can be marked as mutually exclusive using `:mutex(argument_or_option, ...)` method of the Parser class.

```
parser:mutex(
   parser:argument "input"
      :args "?",
   parser:flag "--process-stdin"
)

parser:mutex(
   parser:flag "-q --quiet",
   parser:flag "-v --verbose"
)
```

If more than one element of a mutually exclusive group is used, an error is raised.

```
$ lua script.lua -qv
```

```
Usage: script.lua ([-q] | [-v]) [-h] ([<input>] | [--process-stdin])

Error: option '-v' can not be used together with option '-q'
```

```
$ lua script.lua file --process-stdin
```

```
Usage: script.lua ([-q] | [-v]) [-h] ([<input>] | [--process-stdin])

Error: option '--process-stdin' can not be used together with argument 'input'
```

CHAPTER 5

Adding and configuring commands

A command is a subparser invoked when its name is passed as an argument. For example, in git CLI `add`, `commit`, `push`, etc. are commands. Each command has its own set of arguments and options, but inherits options of its parent.

Commands can be added using `:command(name, description, epilog)` method. Just as options, commands can have several aliases.

```
parser:command "install i"
```

If a command it used, `true` is stored in the corresponding field of the result table.

```
$ lua script.lua install
```

```
{
   install = true
}
```

A typo will result in an appropriate error message.

```
$ lua script.lua instal
```

```
Usage: script.lua [-h] <command> ...

Error: unknown command 'instal'
Did you mean 'install'?
```

5.1 Getting name of selected command

Use `command_target` property of the parser to store the name of used command in a field of the result table.

```
parser:command_target("command")
parser:command("install")
parser:command("remove")
```

```
$ lua script.lua install
```

```
{
   install = true,
   command = "install"
}
```

5.2 Adding elements to commands

The Command class is a subclass of the Parser class, so all the Parser's methods for adding elements work on commands, too.

```
local install = parser:command "install"
install:argument "rock"
install:option "-f --from"
```

```
$ lua script.lua install foo --from=bar
```

```
{
   install = true,
   rock = "foo",
   from = "bar"
}
```

Commands have their own usage and help messages.

```
$ lua script.lua install
```

```
Usage: script.lua install [-h] [-f <from>] <rock>

Error: too few arguments
```

```
$ lua script.lua install --help
```

```
Usage: script.lua install [-h] [-f <from>] <rock>

Arguments:
   rock

Options:
   -h, --help            Show this help message and exit.
   -f <from>, --from <from>
```

5.3 Making a command optional

By default, if a parser has commands, using one of them is obligatory.

```
local parser = argparse()
parser:command "install"
```

```
$ lua script.lua
```

```
Usage: script.lua [-h] <command> ...

Error: a command is required
```

This can be changed using `require_command` property.

```
local parser = argparse()
   :require_command(false)
parser:command "install"
```

5.4 Command summaries

The description for commands shown in the parent parser help message can be set with the `summary` property.

```
parser:command "install"
   :summary "Install a rock."
   :description "A long description for the install command."
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] <command> ...

Options:
   -h, --help            Show this help message and exit.

Commands:
   install               Install a rock.
```

```
$ lua script.lua install --help
```

```
Usage: script.lua install [-h]

A long description for the install command.

Options:
   -h, --help            Show this help message and exit.
```

CHAPTER 6

Default values

For elements such as arguments and options, if `default` property is set to a string, its value is stored in case the element was not used (if it's not a string, it'll be used as `init` property instead, see *Argument and option actions*).

```
parser:option("-o --output", "Output file.", "a.out")
-- Equivalent:
parser:option "-o" "--output"
   :description "Output file."
   :default "a.out"
```

```
$ lua script.lua
```

```
{
   output = "a.out"
}
```

The existence of a default value is reflected in help message, unless `show_default` property is set to `false`.

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [-o <output>]

Options:
   -h, --help            Show this help message and exit.
   -o <output>, --output <output>
                         Output file. (default: a.out)
```

Note that invocation without required arguments is still an error.

```
$ lua script.lua -o
```

```
Usage: script.lua [-h] [-o <output>]

Error: too few arguments
```

6.1 Default mode

`defmode` property regulates how argparse should use the default value of an element.

By default, or if `defmode` contains `u` (for unused), the default value will be automatically passed to the element if it was not invoked at all. It will be passed minimal required of times, so that if the element is allowed to consume no arguments (e.g. using `:args "?"`), the default value is ignored.

If `defmode` contains `a` (for argument), the default value will be automatically passed to the element if not enough arguments were passed, or not enough invocations were made.

Consider the difference:

```
parser:option "-o"
   :default "a.out"
parser:option "-p"
   :default "password"
   :defmode "arg"
```

```
$ lua script.lua -h
```

```
Usage: script.lua [-h] [-o <o>] [-p [<p>]]

Options:
   -h, --help            Show this help message and exit.
   -o <o>                default: a.out
   -p [<p>]              default: password
```

```
$ lua script.lua
```

```
{
   o = "a.out"
}
```

```
$ lua script.lua -p
```

```
{
   o = "a.out",
   p = "password"
}
```

```
$ lua script.lua -o
```

```
Usage: script.lua [-h] [-o <o>] [-p [<p>]]

Error: too few arguments
```

CHAPTER 7

Callbacks

7.1 Converters

argparse can perform automatic validation and conversion on arguments. If `convert` property of an element is a function, it will be applied to all the arguments passed to it. The function should return `nil` and, optionally, an error message if conversion failed. Standard `tonumber` and `io.open` functions work exactly like that.

```
parser:argument "input"
   :convert(io.open)
parser:option "-t --times"
   :convert(tonumber)
```

```
$ lua script.lua foo.txt -t5
```

```
{
   input = file_object,
   times = 5
}
```

```
$ lua script.lua nonexistent.txt
```

```
Usage: script.lua [-h] [-t <times>] <input>

Error: nonexistent.txt: No such file or directory
```

```
$ lua script.lua foo.txt --times=many
```

```
Usage: script.lua [-h] [-t <times>] <input>

Error: malformed argument 'many'
```

If `convert` property of an element is an array of functions, they will be used as converters for corresponding arguments in case the element accepts multiple arguments.

```
parser:option "--line-style"
   :args(2)
   :convert {string.lower, tonumber}
```

```
$ lua script.lua --line-style DASHED 1.5
```

```
{
   line_style = {"dashed", 1.5}
}
```

7.1.1 Table converters

If convert property of an element is a table and doesn't have functions in array part, arguments passed to it will be used as keys. If a key is missing, an error is raised.

```
parser:argument "choice"
   :convert {
      foo = "Something foo-related",
      bar = "Something bar-related"
   }
```

```
$ lua script.lua bar
```

```
{
   choice = "Something bar-related"
}
```

```
$ lua script.lua baz
```

```
Usage: script.lua [-h] <choice>

Error: malformed argument 'baz'
```

7.2 Actions

7.2.1 Argument and option actions

argparse uses action callbacks to process invocations of arguments and options. Default actions simply put passed arguments into the result table as a single value or insert into an array depending on number of arguments the option can take and how many times it can be used.

A custom action can be set using `action` property. An action must be a function. and will be called after each invocation of the option or the argument it is assigned to. Four arguments are passed: result table, target index in that table, an argument or an array of arguments passed by user, and overwrite flag used when an option is invoked too many times.

Converters are applied before actions.

Initial value to be stored at target index in the result table can be set using `init` property, or also using `default` property if the value is not a string.

```
parser:option("--exceptions"):args("*"):action(function(args, _, exceptions)
   for _, exception in ipairs(exceptions) do
      table.insert(args.exceptions, exception)
   end
end):init({"foo", "bar"})

parser:flag("--no-exceptions"):action(function(args)
   args.exceptions = {}
end)
```

```
$ lua script.lua --exceptions x y --exceptions z t
```

```
{
   exceptions = {
      "foo",
      "bar",
      "x",
      "y",
      "z",
      "t"
   }
}
```

```
$ lua script.lua --exceptions x y --no-exceptions
```

```
{
   exceptions = {}
}
```

Actions can also be used when a flag needs to print some message and exit without parsing remaining arguments.

```
parser:flag("-v --version"):action(function()
   print("script v1.0.0")
   os.exit(0)
end)
```

```
$ lua script.lua -v
```

```
script v1.0.0
```

7.2.2 Built-in actions

These actions can be referred to by their string names when setting `action` property:

Name	Description
store	Stores argument or arguments at target index.
store_true	Stores `true` at target index.
store_false	Stores `false` at target index.
count	Increments number at target index.
append	Appends argument or arguments to table at target index.
concat	Appends arguments one by one to table at target index.

Examples using `store_false` and `concat` actions:

```
parser:flag("--candy")
parser:flag("--no-candy"):target("candy"):action("store_false")
parser:flag("--rain", "Enable rain", false)
parser:option("--exceptions"):args("*"):action("concat"):init({"foo", "bar"})
```

```
$ lua script.lua
```

```
{
   rain = false
}
```

```
$ lua script.lua --candy
```

```
{
   candy = true,
   rain = false
}
```

```
$ lua script.lua --no-candy --rain
```

```
{
   candy = false,
   rain = true
}
```

```
$ lua script.lua --exceptions x y --exceptions z t
```

```
{
   exceptions = {
      "foo",
      "bar",
      "x",
      "y",
      "z",
      "t"
   },
   rain = false
}
```

7.2.3 Command actions

Actions for parsers and commands are simply callbacks invoked after parsing, with result table and command name as the arguments. Actions for nested commands are called first.

```
local install = parser:command("install"):action(function(args, name)
   print("Running " .. name)
   -- Use args here
)

parser:action(function(args)
   print("Callbacks are fun!")
end)
```

```
$ lua script.lua install
```

```
Running install
Callbacks are fun!
```

CHAPTER 8

Configuring help and usage messages

The usage and help messages of parsers and commands can be generated on demand using `:get_usage()` and `:get_help()` methods, and overridden using `help` and `usage` properties. When using the autogenerated usage and help messages, there are several ways to adjust them.

8.1 Hiding arguments, options, and commands from messages

If `hidden` property for an argument, an option or a command is set to `true`, it is not included into help and usage messages, but otherwise works normally.

```
parser:option "--normal-option"
parser:option "--deprecated-option"
   :hidden(true)
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [--normal-option <normal_option>]

Options:
   -h, --help            Show this help message and exit.
   --normal-option <normal_option>
```

```
$ lua script.lua --deprecated-option value
```

```
{
   deprecated_option = "value"
}
```

8.2 Hiding option and command aliases

Hidden aliases can be added to an option or command by setting the `hidden_name` property. Its value is interpreted in the same way as the `name` property.

```
parser:option "--server"
   :hidden_name "--from"
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [--server <server>]

Options:
   -h, --help            Show this help message and exit.
   --server <server>
```

```
$ lua script.lua --server foo
$ lua script.lua --from foo
```

```
{
   server = "foo"
}
```

8.3 Setting argument placeholder

For options and arguments, `argname` property controls the placeholder for the argument in the usage message.

```
parser:option "-f" "--from"
   :argname "<server>"
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [-f <server>]

Options:
   -h, --help            Show this help message and exit.
   -f <server>, --from <server>
```

`argname` can be an array of placeholders.

```
parser:option "--pair"
   :args(2)
   :argname {"<key>", "<value>"}
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [--pair <key> <value>]

Options:
   -h, --help            Show this help message and exit.
   --pair <key> <value>
```

8.4 Grouping elements

`:group(name, ...)` method of parsers and commands puts passed arguments, options, and commands into a named group with its own section in the help message. Elements outside any groups are put into a default section.

```
parser:group("Configuring output format",
   parser:flag "-v --verbose",
   parser:flag "--use-colors",
   parser:option "--encoding"
)

parser:group("Configuring processing",
   parser:option "--compression-level",
   parser:flag "--skip-broken-chunks"
)

parser:flag "--version"
   :action(function() print("script.lua 1.0.0") os.exit(0) end)
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [-v] [--use-colors] [--encoding <encoding>]
       [--compression-level <compression_level>]
       [--skip-broken-chunks] [--version]

Configuring output format:
   -v, --verbose
   --use-colors
   --encoding <encoding>

Configuring processing:
   --compression-level <compression_level>
   --skip-broken-chunks

Other options:
   -h, --help            Show this help message and exit.
   --version
```

8.5 Help message line wrapping

If `help_max_width` property of a parser or a command is set, when generating its help message, argparse will automatically wrap lines, attempting to fit into given number of columns. This includes wrapping lines in parser description and epilog and descriptions of arguments, options, and commands.

Line wrapping preserves existing line endings and only splits overly long input lines. When breaking a long line, it replicates indentation of the line in the continuation lines. Additionally, if the first non-space token in a line is `*`, `+`, or `-`, the line is considered a list item, and the continuation lines are aligned with the first word after the list item marker.

```
parser:help_max_width(80)

parser:option "-f --foo"
   :description("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do␣
↪eiusmod tempor " ..
      "incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis␣
↪nostrud exercitation " ..
```

(continues on next page)

(continued from previous page)

```
      "ullamco laboris nisi ut aliquip ex ea commodo consequat.\n" ..
      "The next paragraph is indented:\n" ..
      "  Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore␣
↪eu fugiat nulla pariatur. " ..
      "Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia␣
↪deserunt mollit anim id est laborum.")

parser:option "-b --bar"
   :description("Here is a list:\n"..
      "* Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod␣
↪tempor...\n" ..
      "* Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut␣
↪aliquip...\n" ..
      "* Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore␣
↪eu fugiat nulla pariatur.")
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [-f <foo>] [-b <bar>]

Options:
   -h, --help            Show this help message and exit.
      -f <foo>,          Lorem ipsum dolor sit amet, consectetur adipiscing
   --foo <foo>           elit, sed do eiusmod tempor incididunt ut labore et
                         dolore magna aliqua. Ut enim ad minim veniam, quis
                         nostrud exercitation ullamco laboris nisi ut aliquip ex
                         ea commodo consequat.
                         The next paragraph is indented:
                           Duis aute irure dolor in reprehenderit in voluptate
                           velit esse cillum dolore eu fugiat nulla pariatur.
                           Excepteur sint occaecat cupidatat non proident, sunt
                           in culpa qui officia deserunt mollit anim id est
                           laborum.
      -b <bar>,          Here is a list:
   --bar <bar>           * Lorem ipsum dolor sit amet, consectetur adipiscing
                           elit, sed do eiusmod tempor...
                         * Ut enim ad minim veniam, quis nostrud exercitation
                           ullamco laboris nisi ut aliquip...
                         * Duis aute irure dolor in reprehenderit in voluptate
                           velit esse cillum dolore eu fugiat nulla pariatur.
```

`help_max_width` property is inherited by commands.

8.6 Configuring help and usage message layout

Several other parser and command properties can be used to tweak help and usage message format. Like `help_max_width`, all of them are inherited by commands when set on the parser or a parent command.

`usage_max_width` property sets maximum width of the usage string. Default is `70`.

`usage_margin` property sets margin width used when line wrapping long usage strings. Default is `7`.

```
parser:usage_max_width(50)
   :usage_margin(#"Usage: script.lua ")
```

(continues on next page)

(continued from previous page)

```
parser:option "--foo"
parser:option "--bar"
parser:option "--baz"
parser:option "--qux"

print(parser:get_usage())
```

```
$ lua script.lua
```

```
Usage: script.lua [-h] [--foo <foo>] [--bar <bar>]
                  [--baz <baz>] [--qux <qux>]
```

Help message for a group of arguments, options, or commands is organized into two columns, with usage template on the left side and descriptions on the right side. `help_usage_margin` property sets horizontal offset for the first column (`3` by default). `help_description_margin` property sets horizontal offset for the second column (`25` by default).

`help_vertical_space` property sets number of extra empty lines to put between descriptions for different elements within a group (`0` by default).

```
parser:help_usage_margin(2)
   :help_description_margin(17)
   :help_vertical_space(1)

parser:option("--foo", "Set foo.")
parser:option("--bar", "Set bar.")
parser:option("--baz", "Set baz.")
parser:option("--qux", "Set qux.")
```

```
$ lua script.lua --help
```

```
Usage: script.lua [-h] [--foo <foo>] [--bar <bar>] [--baz <baz>]
       [--qux <qux>]

Options:

  -h, --help     Show this help message and exit.

  --foo <foo>    Set foo.

  --bar <bar>    Set bar.

  --baz <baz>    Set baz.

  --qux <qux>    Set qux.
```

CHAPTER 9

Shell completions

Argparse can generate shell completion scripts for Bash, Zsh, and Fish. The completion scripts support completing options, commands, and argument choices.

The Parser methods `:get_bash_complete()`, `:get_zsh_complete()`, and `:get_fish_complete()` return completion scripts as a string.

9.1 Adding a completion option or command

A `--completion` option can be added to a parser using the `:add_complete([value])` method. The optional `value` argument is a string or table used to configure the option (by calling the option with `value`).

```
local parser = argparse()
   :add_complete()
```

```
$ lua script.lua -h
```

```
Usage: script.lua [-h] [--completion {bash,zsh,fish}]

Options:
   -h, --help            Show this help message and exit.
   --completion {bash,zsh,fish}
                         Output a shell completion script for the specified shell.
```

A similar `completion` command can be added to a parser using the `:add_complete_command([value])` method.

9.2 Using completions

9.2.1 Bash

Save the generated completion script at `/usr/share/bash-completion/completions/script.lua` or `~/.local/share/bash-completion/completions/script.lua`.

Alternatively, add the following line to the `~/.bashrc`:

```
source <(script.lua --completion bash)
```

9.2.2 Zsh

Save the completion script in the `/usr/share/zsh/site-functions/` directory or any directory in the `$fpath`. The file name should be an underscore followed by the program name. A new directory can be added to to the `$fpath` by adding e.g. `fpath=(~/.zfunc $fpath)` in the `~/.zshrc` before `compinit`.

9.2.3 Fish

Save the completion script at `/usr/share/fish/vendor_completions.d/script.lua.fish` or `~/.config/fish/completions/script.lua.fish`.

Alternatively, add the following line to the file `~/.config/fish/config.fish`:

```
script.lua --completion fish | source
```

CHAPTER 10

Miscellaneous

10.1 Argparse version

`argparse` module is a table with `__call` metamethod. `argparse.version` is a string in `MAJOR.MINOR.PATCH` format specifying argparse version.

10.2 Overwriting default help option

If the property `add_help` of a parser is set to `false`, no help option will be added to it. Otherwise, the value of the field will be used to configure it.

```
local parser = argparse()
   :add_help "/?"
```

```
$ lua script.lua /?
```

```
Usage: script.lua [/?]

Options:
   /?                    Show this help message and exit.
```

10.3 Help command

A help command can be added to the parser using the `:add_help_command([value])` method. It accepts an optional string or table value which is used to configure the command.

```
local parser = argparse()
   :add_help_command()
```

(continues on next page)

(continued from previous page)

```
parser:command "install"
   :description "Install a rock."
```

```
$ lua script.lua help
```

```
Usage: script.lua [-h] <command> ...

Options:
   -h, --help            Show this help message and exit.

Commands:
   help                  Show help for commands.
   install               Install a rock.
```

```
$ lua script.lua help install
```

```
Usage: script.lua install [-h]

Install a rock.

Options:
   -h, --help            Show this help message and exit.
```

10.4 Disabling option handling

When `handle_options` property of a parser or a command is set to `false`, all options will be passed verbatim to the argument list, as if the input included double-hyphens.

```
parser:handle_options(false)
parser:argument "input"
   :args "*"
parser:option "-f" "--foo"
   :args "*"
```

```
$ lua script.lua bar -f --foo bar
```

```
{
   input = {"bar", "-f", "--foo", "bar"}
}
```

10.5 Prohibiting overuse of options

By default, if an option is invoked too many times, latest invocations overwrite the data passed earlier.

```
parser:option "-o --output"
```

```
$ lua script.lua -oFOO -oBAR
```

```
{
   output = "BAR"
}
```

Set `overwrite` property to `false` to prohibit this behavior.

```
parser:option "-o --output"
   :overwrite(false)
```

```
$ lua script.lua -oFOO -oBAR
```

```
Usage: script.lua [-h] [-o <output>]

Error: option '-o' must be used at most 1 time
```

10.6 Parsing algorithm

argparse interprets command line arguments in the following way:

Argument	Interpretation
`foo`	An argument of an option or a positional argument.
`--foo`	An option.
`--foo=bar`	An option and its argument. The option must be able to take arguments.
`-f`	An option.
`-abcdef`	Letters are interpreted as options. If one of them can take an argument, the rest of the string is passed to it.
`--`	The rest of the command line arguments will be interpreted as positional arguments.

10.7 Property lists

10.7.1 Parser properties

Properties that can be set as arguments when calling or constructing a parser, in this order:

Property	Type
`name`	String
`description`	String
`epilog`	String

Other properties:

Property	Type
usage	String
help	String
require_command	Boolean
handle_options	Boolean
add_help	Boolean or string or table
command_target	String
usage_max_width	Number
usage_margin	Number
help_max_width	Number
help_usage_margin	Number
help_description_margin	Number
help_vertical_space	Number

10.7.2 Command properties

Properties that can be set as arguments when calling or constructing a command, in this order:

Property	Type
name	String
description	String
epilog	String

Other properties:

Property	Type
hidden_name	String
summary	String
target	String
usage	String
help	String
require_command	Boolean
handle_options	Boolean
action	Function
add_help	Boolean or string or table
command_target	String
hidden	Boolean
usage_max_width	Number
usage_margin	Number
help_max_width	Number
help_usage_margin	Number
help_description_margin	Number
help_vertical_space	Number

10.7.3 Argument properties

Properties that can be set as arguments when calling or constructing an argument, in this order:

Property	Type
`name`	String
`description`	String
`default`	Any
`convert`	Function or table
`args`	Number or string

Other properties:

Property	Type
`target`	String
`defmode`	String
`show_default`	Boolean
`argname`	String or table
`choices`	Table
`action`	Function or string
`init`	Any
`hidden`	Boolean

10.7.4 Option and flag properties

Properties that can be set as arguments when calling or constructing an option or a flag, in this order:

Property	Type
`name`	String
`description`	String
`default`	Any
`convert`	Function or table
`args`	Number or string
`count`	Number or string

Other properties:

Property	Type
`hidden_name`	String
`target`	String
`defmode`	String
`show_default`	Boolean
`overwrite`	Booleans
`argname`	String or table
`choices`	Table
`action`	Function or string
`init`	Any
`hidden`	Boolean

This is a tutorial for argparse, a feature-rich command line parser for Lua.