Cleo Documentation

Release 0.6.1

Sébastien Eustace

Dec 08, 2018

Contents

Cleo allows you to create beautiful and testable command-line commands.

It is heavily inspired by the Symfony Console Component, with some useful additions.

CHAPTER 1

Installation

You can install Cleo in various ways:

- Using [Poetry](https://poetry.eustace.io):

```
$ poetry add cleo
```

- Using `pip`

```
$ pip install cleo
```

- Use the official repository (https://github.com/sdispater/cleo)

CHAPTER 2

Usage

To make a command that greets you from the command line, create `greet_command.py` and add the following to it:

```
from cleo import Command


class GreetCommand(Command):
    """
    Greets someone

    greet
        {name? : Who do you want to greet?}
        {--y|yell : If set, the task will yell in uppercase letters}
    """

    def handle(self):
        name = self.argument('name')

        if name:
            text = 'Hello {}'.format(name)
        else:
            text = 'Hello'

        if self.option('yell'):
            text = text.upper()

        self.line(text)
```

You also need to create the file to run at the command line which creates an `Application` and adds commands to it:

```
#!/usr/bin/env python

from greet_command import GreetCommand
```

(continues on next page)

(continued from previous page)

```
from cleo import Application

application = Application()
application.add(GreetCommand())

if __name__ == '__main__':
    application.run()
```

Test the new command by running the following

```
$ python application.py greet John
```

This will print the following to the command line:

```
Hello John
```

You can also use the `--yell` option to make everything uppercase:

```
$ python application.py greet John --yell
```

This prints:

```
HELLO JOHN
```

As you may have already seen, Cleo uses the command docstring to determine the command definition. The docstring must be in the following form :

```
"""
Command description

Command signature
"""
```

The signature being in the following form:

```
"""
command:name {argument : Argument description} {--option : Option description}
"""
```

The signature can span multiple lines.

```
"""
command:name
    {argument : Argument description}
    {--option : Option description}
"""
```

2.1 Coloring the Output

Whenever you output text, you can surround the text with tags to color its output. For example:

```
# green text
self.line('<info>foo</info>')
```

(continues on next page)

(continued from previous page)

```
# yellow text
self.line('<comment>foo</comment>')

# black text on a cyan background
self.line('<question>foo</question>')

# white text on a red background
self.line('<error>foo</error>')
```

The closing tag can be replaced by `</>`, which revokes all formatting options established by the last opened tag.

It is possible to define your own styles using the `add_style()` method:

```
self.add_style('fire', fg='red', bg='yellow', options=['bold', 'blink'])
self.line('<fire>foo</fire>')
```

Available foreground and background colors are: `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`.

And available options are: `bold`, `underscore`, `blink`, `reverse` and `conceal`.

You can also set these colors and options inside the tag name:

```
# green text
self.line('<fg=green>foo</>')

# black text on a cyan background
self.line('<fg=black;bg=cyan>foo</>')

# bold text on a yellow background
self.line('<bg=yellow;options=bold>foo</>')
```

2.2 Verbosity Levels

Cleo has four verbosity levels. These are defined in the `Output` class:

Mode	Meaning	Console option
`NA`	Do not output any messages	`-q` or `--quiet`
`clikit.VERBOSITY_NORMAL`	The default verbosity level	(none)
`clikit.VERBOSITY_VERBOSE`	Increased verbosity of messages	`-v`
`clikit.VERBOSITY_VERY_VERBOSE`	Informative non essential messages	`-vv`
`clikit.VERBOSITY_DEBUG`	Debug messages	`-vvv`

It is possible to print a message in a command for only a specific verbosity level. For example:

```
if clikit.VERBOSITY_VERBOSE <= self.io.verbosity:
    self.line(...)
```

There are also more semantic methods you can use to test for each of the verbosity levels:

```
if self.output.is_quiet():
    # ...
```

(continues on next page)

(continued from previous page)

```
if self.output.is_verbose():
    # ...
```

You can also pass the verbosity flag directly to *line()*.

```
self.line("", verbosity=clikit.VERBOSITY_VERBOSE)
```

When the quiet level is used, all output is suppressed.

2.3 Using Arguments

The most interesting part of the commands are the arguments and options that you can make available. Arguments are the strings - separated by spaces - that come after the command name itself. They are ordered, and can be optional or required. For example, add an optional `last_name` argument to the command and make the `name` argument required:

```
class GreetCommand(Command):
    """
    Greets someone

    greet
        {name : Who do you want to greet?}
        {last_name? : Your last name?}
        {--y|yell : If set, the task will yell in uppercase letters}
    """
```

You now have access to a `last_name` argument in your command:

```
last_name = self.argument('last_name')
if last_name:
    text += ' {}'.format(last_name)
```

The command can now be used in either of the following ways:

```
$ python application.py greet John
$ python application.py greet John Doe
```

It is also possible to let an argument take a list of values (imagine you want to greet all your friends). For this it must be specified at the end of the argument list:

```
class GreetCommand(Command):
    """
    Greets someone

    greet
        {names* : Who do you want to greet?}
        {--y|yell : If set, the task will yell in uppercase letters}
    """
```

To use this, just specify as many names as you want:

```
$ python application.py demo:greet John Jane
```

You can access the `names` argument as a list:

```
names = self.argument('names')
if names:
    text += ' {}'.format(', '.join(names))
```

There are 3 argument variants you can use:

Mode	Notation	Value
`clikit.ARGUMENT_REQUIRED`	none (just write the argument name)	The argument is required
`clikit.ARGUMENT_OPTIONAL`	`argument?`	The argument is optional and therefore can be omitted
`clikit.ARGUMENT_MULTI_VALUED`	`argument*`	The argument can contain an indefinite number of arguments and must be used at the end of the argument list

You can combine them like this:

```
class GreetCommand(Command):
    """
    Greets someone

    greet
        {names?* : Who do you want to greet?}
        {--y|yell : If set, the task will yell in uppercase letters}
    """
```

If you want to set a default value, you can it like so:

```
argument=default
```

The argument will then be considered optional.

2.4 Using Options

Unlike arguments, options are not ordered (meaning you can specify them in any order) and are specified with two dashes (e.g. `--yell` - you can also declare a one-letter shortcut that you can call with a single dash like `-y`). Options are *always* optional, and can be setup to accept a value (e.g. `--dir=src`) or simply as a boolean flag without a value (e.g. `--yell`).

Tip: It is also possible to make an option *optionally* accept a value (so that `--yell` or `--yell=loud` work). Options can also be configured to accept a list of values.

For example, add a new option to the command that can be used to specify how many times in a row the message should be printed:

```
class GreetCommand(Command):
    """
    Greets someone

    greet
        {name? : Who do you want to greet?}
        {--y|yell : If set, the task will yell in uppercase letters}
```

(continues on next page)

(continued from previous page)

```
        {--iterations=1 : How many times should the message be printed?}
    """
```

Next, use this in the command to print the message multiple times:

```
for _ in range(0, self.option('iterations')):
    self.line(text)
```

Now, when you run the task, you can optionally specify a `--iterations` flag:

```
$ python application.py demo:greet John
$ python application.py demo:greet John --iterations=5
```

The first example will only print once, since `iterations` is empty and defaults to `1`. The second example will print five times.

Recall that options don't care about their order. So, either of the following will work:

```
$ python application.py demo:greet John --iterations=5 --yell
$ python application.py demo:greet John --yell --iterations=5
```

There are 4 option variants you can use:

Option	Notation	Value
`clikit.OPTION_MULTI_VALUED`	`--option=*`	This option accepts multiple values (e.g. `--dir=/foo --dir=/bar`)
`clikit.OPTION_NO_VALUE`	`--option`	Do not accept input for this option (e.g. `--yell`)
`clikit.OPTION_REQUIRED_VALUE`	`--option=`	This value is required (e.g. `--iterations=5`), the option itself is still optional
`clikit.OPTION_OPTIONAL_VALUE`	`--option=?`	This option may or may not have a value (e.g. `--yell` or `--yell=loud`)

You can combine them like this:

```
class GreetCommand(Command):
    """
    Greets someone

    greet
        {name? : Who do you want to greet?}
        {--y|yell : If set, the task will yell in uppercase letters}
        {--iterations=?*1 : How many times should the message be printed?}
    """
```

2.5 Helpers

Cleo also contains a set of "helpers" - different small tools capable of helping you with different tasks:

- *Question Helper*: interactively ask the user for information
- *Progress Bar*: shows a progress bar
- *Table*: displays tabular data as a table

2.6 Testing Commands

Cleo provides several tools to help you test your commands. The most useful one is the `CommandTester` class. It uses a special IO class to ease testing without a real console:

```
import pytest

from cleo import Application
from cleo import CommandTester

def test_execute(self):
    application = Application()
    application.add(GreetCommand())

    command = application.find('demo:greet')
    command_tester = CommandTester(command)
    command_tester.execute()

    assert "..." == tester.io.fetch_output()
```

The `CommandTester.io.fetch_output()` method returns what would have been displayed during a normal call from the console. `CommandTester.io.fetch_error()` is also available to get what you have been written to the stderr.

You can test sending arguments and options to the command by passing them as a string to the `CommandTester.execute()` method:

```
import pytest

from cleo import Application
from cleo import CommandTester

def test_execute(self):
    application = Application()
    application.add(GreetCommand())

    command = application.find('demo:greet')
    command_tester = CommandTester(command)
    command_tester.execute("John")

    assert "John" in tester.io.fetch_output()
```

2.6.1 Testing with user inputs

To test user inputs, you pass it to `execute()`.

```
command_tester = CommandTester(command)
command_tester.execute(inputs="123\nfoo\nbar")
```

Tip: You can also test a whole console application by using the `ApplicationTester` class.

2.7 Calling an existing Command

If a command depends on another one being run before it, instead of asking the user to remember the order of execution, you can call it directly yourself. This is also useful if you want to create a "meta" command that just runs a bunch of other commands.

Calling a command from another one is straightforward:

```
def handle(self):
    return_code = self.call('demo:greet', "John --yell")

    # ...
```

Tip: If you want to suppress the output of the executed command, you can use the `call_silent()` method instead.

2.8 Overwrite the current line

If you want to overwrite the current line, you can use the `overwrite()` method.

```
def handle(self):
    self.write('Processing...')
    # do some work
    self.overwrite('Done!')
```

Warning: `overwrite()` will only work in combination with the `write()` method which does not add a new line.

Note: `overwrite()` does not automatically add a new line so you must call `line('')` if necessary.

2.9 Autocompletion

Cleo supports automatic (tab) completion in `bash`, `zsh` and `fish`.

You can register completion for your application by running one of the following in a terminal, replacing `[program]` with the command you use to run your application:

```
# BASH - Ubuntu / Debian
[program] completions bash | sudo tee /etc/bash_completion.d/[program].bash-completion

# BASH - Mac OSX (with Homebrew "bash-completion")
[program] completions bash > $(brew --prefix)/etc/bash_completion.d/[program].bash-
↪completion

# ZSH - Config file
mkdir ~/.zfunc
echo "fpath+=~/.zfunc" >> ~/.zshrc
```

(continues on next page)

(continued from previous page)

```
[program] completions zsh > ~/.zfunc/_test

# FISH
[program] completions fish > ~/.config/fish/completions/[program].fish
```

CHAPTER 3

Helpers

Cleo also contains a set of "helpers" - different small tools capable of helping you with different tasks:

3.1 Question Helper

3.1.1 Asking the User for Confirmation

Suppose you want to confirm an action before actually executing it. Add the following to your command:

```
def handle(self):
    if not self.confirm('Continue with this action?', False):
        return
```

In this case, the user will be asked "Continue with this action?". If the user answers with `y` it returns `True` or `False` if they answer with `n`. The second argument to `confirm()` is the default value to return if the user doesn't enter any valid input. If the second argument is not provided, `True` is assumed.

Tip: You can customize the regex used to check if the answer means "yes" in the third argument of the `customize()` method. For instance, to allow anything that starts with either `y` or `j`, you would set it to:

```
self.confirm('Continue with this action?', False, '(?i)^(y|j)')
```

The regex defaults to `(?i)^y`.

3.1.2 Asking the User for Information

You can also ask a question with more than a simple yes/no answer. For instance, if you want to know a user name, you can add this to your command:

```
def handle(self):
    name = self.ask('Please enter your name', 'John Doe')
```

The user will be asked "Please enter your name". They can type some name which will be returned by the `ask()` method. If they leave it empty, the default value (`John Doe` here) is returned.

Let the User Choose from a List of Answers

If you have a predefined set of answers the user can choose from, you could use a `ChoiceQuestion` or the `choice()` method which makes sure that the user can only enter a valid string from a predefined list:

```
def handle(self):
    color = self.choice(
        'Please select your favorite color (defaults to red)',
        ['red', 'blue', 'yellow'],
        0
    )

    self.line('You have just selected: %s' % color)
```

The option which should be selected by default is provided with the third argument. The default is `None`, which means that no option is the default one.

If the user enters an invalid string, an error message is shown and the user is asked to provide the answer another time, until they enter a valid string or reach the maximum number of attempts. The default value for the maximum number of attempts is `None`, which means infinite number of attempts.

Multiple Choices

Sometimes, multiple answers can be given. The `ChoiceQuestion` or `choice()` method provides this feature using comma separated values. This is disabled by default, to enable this use the `multiple` keyword if using the `choice()` method or the `multiselect` attribute if using the `ChoiceQuestion` directly:

```
def handle(self):
    colors = self.choice(
        'Please select your favorite color (defaults to red and blue),
        ['red', 'blue', 'yellow'],
        '0,1'
        multiple=True
    )

    self.line('You have just selected: %s' % ', '.join(colors))
```

Now, when the user enters `1,2`, the result will be: `You have just selected:  blue, yellow`.

If the user does not enter anything, the result will be: `You have just selected:  red, blue`.

Autocompletion

You can also specify an array of potential answers for a given question. These will be autocompleted as the user types:

```
def handle(self):
    names = ['John', 'Jane', 'Paul']
    question = self.create_question('Please enter a name', default='John')
```

(continues on next page)

(continued from previous page)

```
    question.set_autocomplete_values(names)

    name = self.ask(question)
```

Hiding the User's Response

You can also ask a question and hide the response. This is particularly convenient for passwords:

```
def handle(self):
    password = self.secret('What is the database password?')
```

3.1.3 Validating the Answer

You can even validate the answer. For instance, you might only accept integers:

```
def handle(self):
    question = self.create_question('Choose a number')
    question.set_validator(int)
    question.set_max_attempts(2)

    number = self.ask(question)
```

The `validator` a callback which handles the validation. It should throw an exception if there is something wrong. The exception message is displayed in the console, so it is a good practice to put some useful information in it. The validator or the callback function should also return the value of the user's input if the validation was successful.

You can set the max number of times to ask with the `set_max_attempts()` method. If you reach this max number it will use the default value. Using `None` means the amount of attempts is infinite. The user will be asked as long as they provide an invalid answer and will only be able to proceed if their input is valid.

3.1.4 Testing a Command that Expects Input

If you want to write a unit test for a command which expects some kind of input from the command line, you need to set the helper input stream:

```
def test_execute_command(self):
    command_tester = CommandTester(command)
    # Equals to a user inputting "Test" and hitting ENTER
    # If you need to enter a confirmation, "yes\n" will work

    command_tester.execute(inputs="Test\n")
```

3.2 Progress Bar

When executing longer-running commands, it may be helpful to show progress information, which updates as your command runs:

To display progress details, use the `progress_bar()` method (which returns a `ProgressBar` instance), pass it a total number of units, and advance the progress as the command executes:

```
def handle(self):
    # Create a new progress bar (50 units)
    progress = self.progress_bar(50)

    # Start and displays the progress bar
    for _ in range(50):
        # ... do some work

        # Advance the progress bar 1 unit
        progress.advance()

        # You can also advance the progress bar by more than 1 unit
        # progress.advance(3)

    # Ensure that the progress bar is at 100%
    progress.finish()
```

Instead of advancing the bar by a number of steps (with the `advance()` method), you can also set the current progress by calling the `set_progress()` method.

Tip: If your platform doesn't support ANSI codes, updates to the progress bar are added as new lines. To prevent the output from being flooded, adjust the `set_redraw_frequency()` accordingly. By default, when using a `max`, the redraw frequency is set to *10%* of your `max`.

If you don't know the number of steps in advance, just omit the steps argument when using the `progress_bar` method:

```
progress = self.progress_bar()
```

The progress will then be displayed as a throbber:

```
# no max steps (displays it like a throbber)
   0 [>--------------------------]
   5 [----->--------------------]
   5 [============================]

# max steps defined
 0/3 [>--------------------------]   0%
 1/3 [=========>------------------]  33%
 3/3 [============================] 100%
```

Whenever your task is finished, don't forget to call `finish()` to ensure that the progress bar display is refreshed with a 100% completion.

Note: If you want to output something while the progress bar is running, call `clear()` first. After you're done, call `display()` to show the progress bar again.

3.2.1 Customizing the Progress Bar

Built-in Formats

By default, the information rendered on a progress bar depends on the current level of verbosity of the `IO` instance:

```
# clikit.VERBOSITY_NORMAL (CLI with no verbosity flag)
 0/3 [>----------------------------]   0%
 1/3 [=========>------------------]  33%
 3/3 [============================] 100%

# clikit.VERBOSITY_VERBOSE (-v)
 0/3 [>----------------------------]   0%  1 sec
 1/3 [=========>------------------]  33%  1 sec
 3/3 [============================] 100%  1 sec

# clikit.VERBOSITY_VERY_VERBOSE (-vv)
 0/3 [>----------------------------]   0%  1 sec
 1/3 [=========>------------------]  33%  1 sec
 3/3 [============================] 100%  1 sec

# clikit.VERBOSITY_DEBUG (-vvv)
 0/3 [>----------------------------]   0%  1 sec/1 sec  1.0 MB
 1/3 [=========>------------------]  33%  1 sec/1 sec  1.0 MB
 3/3 [============================] 100%  1 sec/1 sec  1.0 MB
```

Note: If you call a command with the quiet flag (`-q`), the progress bar won't be displayed.

Instead of relying on the verbosity mode of the current command, you can also force a format via `set_format()`:

```
progress.set_format('verbose')
```

The built-in formats are the following:

- `normal`
- `verbose`
- `very_verbose`
- `debug`

If you don't set the number of steps for your progress bar, use the `_nomax` variants:

- `normal_nomax`
- `verbose_nomax`
- `very_verbose_nomax`
- `debug_nomax`

Custom Formats

Instead of using the built-in formats, you can also set your own:

```
progress.set_format('%bar%')
```

This sets the format to only display the progress bar itself:

```
>----------------------------
=========>------------------
============================
```

A progress bar format is a string that contains specific placeholders (a name enclosed with the `%` character); the placeholders are replaced based on the current progress of the bar. Here is a list of the built-in placeholders:

- `current`: The current step
- `max`: The maximum number of steps (or 0 if no max is defined)
- `bar`: The bar itself
- `percent`: The percentage of completion (not available if no max is defined)
- `elapsed`: The time elapsed since the start of the progress bar
- `remaining`: The remaining time to complete the task (not available if no max is defined)
- `estimated`: The estimated time to complete the task (not available if no max is defined)
- `memory`: The current memory usage
- `message`: The current message attached to the progress bar

For instance, here is how you could set the format to be the same as the `debug` one:

```
progress.set_format(' %current%/%max% [%bar%] %percent:3s%% %elapsed:6s%/%estimated:-
↪6s% %memory:6s%')
```

Notice the `:6s` part added to some placeholders? That's how you can tweak the appearance of the bar (formatting and alignment). The part after the colon (`:`) is used to set the format of the string.

The `message` placeholder is a bit special as you must set the value yourself:

```
progress.set_message('Task starts')
progress.start()

progress.set_message('Task in progress...')
progress.advance()

# ...

progress.set_message('Task is finished')
progress.finish()
```

Bar Settings

Amongst the placeholders, `bar` is a bit special as all the characters used to display it can be customized:

```
# the finished part of the bar
progress.set_bar_character('<comment>=</comment>')

# the unfinished part of the bar
progress.set_empty_bar_character(' ')

# the progress character
progress.set_progress_character('|')

# the bar width
progress.set_bar_width(50)
```

Warning: For performance reasons, be careful if you set the total number of steps to a high number. For example, if you're iterating over a large number of items, consider setting the redraw frequency to a higher value by calling `ProgressHelper.set_redraw_frequency()`, so it updates on only some iterations:

```
progress.start(50000)

# update every 100 iterations
progress.set_redraw_frequency(100)

for _ in range(50000)
    # ... do some work

    progress.advance()
```

3.3 Table

When building a console application it may be useful to display tabular data:

```
+---------------+--------------------------+------------------+
| ISBN          | Title                    | Author           |
+---------------+--------------------------+------------------+
| 99921-58-10-7 | Divine Comedy            | Dante Alighieri  |
| 9971-5-0210-0 | A Tale of Two Cities     | Charles Dickens  |
| 960-425-059-0 | The Lord of the Rings    | J. R. R. Tolkien |
| 80-902734-1-6 | And Then There Were None | Agatha Christie  |
+---------------+--------------------------+------------------+
```

To display a table, use the `table()` method, set the headers, set the rows and then render the table:

```
def handle(self):
    table = self.table()

    table.set_header_row(['ISBN', 'Title', 'Author'])
    table.set_rows([
        ['99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'],
        ['9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'],
        ['960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'],
        ['80-902734-1-6', 'And Then There Were None', 'Agatha Christie']
    ])

    table.render(self.io)
```

Tip: All these steps can be done in one go using the `render_table` method:

```
self.render_table(
    ['ISBN', 'Title', 'Author'],
    [
        ['99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'],
        ['9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'],
        ['960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'],
        ['80-902734-1-6', 'And Then There Were None', 'Agatha Christie']
    ]
)
```

You can add a table separator anywhere in the output by using `table_seprator()`, which returns a `TableSeparator`, as a row:

```
table.set_rows([
    ['99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'],
    ['9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'],
    self.table_separator(),
    ['960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'],
    ['80-902734-1-6', 'And Then There Were None', 'Agatha Christie']
])
```

```
+---------------+--------------------------+------------------+
| ISBN          | Title                    | Author           |
+---------------+--------------------------+------------------+
| 99921-58-10-7 | Divine Comedy            | Dante Alighieri  |
| 9971-5-0210-0 | A Tale of Two Cities     | Charles Dickens  |
+---------------+--------------------------+------------------+
| 960-425-059-0 | The Lord of the Rings    | J. R. R. Tolkien |
| 80-902734-1-6 | And Then There Were None | Agatha Christie  |
+---------------+--------------------------+------------------+
```

The table style can be changed to any built-in styles via `set_style()`:

```
# same as calling nothing
table.set_style('default')

# changes the default style to compact
table.set_style('compact')
```

This code results in:

```
ISBN          Title                    Author
99921-58-10-7 Divine Comedy            Dante Alighieri
9971-5-0210-0 A Tale of Two Cities     Charles Dickens
960-425-059-0 The Lord of the Rings    J. R. R. Tolkien
80-902734-1-6 And Then There Were None Agatha Christie
```

You can also set the style to `borderless`:

```
table.set_style('borderless')
```

which outputs:

```
=============== ========================== ==================
 ISBN            Title                      Author
=============== ========================== ==================
 99921-58-10-7   Divine Comedy              Dante Alighieri
 9971-5-0210-0   A Tale of Two Cities       Charles Dickens
 960-425-059-0   The Lord of the Rings      J. R. R. Tolkien
 80-902734-1-6   And Then There Were None   Agatha Christie
=============== ========================== ==================
```

If the built-in styles do not fit your need, define your own:

```
# by default, this is based on the default style
style = self.table_style()
```

(continues on next page)

(continued from previous page)

```
# customize the style
style.set_horizontal_border_char('<fg=magenta>|</>')
style.set_vertical_border_char('<fg=magenta>-</>')
style.set_crossing_char(' ')

# use the style for this table
table.set_style(style)
```

Here is a full list of things you can customize:

- `set_adding_char()`
- `set_horizontal_border_char()`
- `set_vertical_border_char()`
- `set_crossing_char()`
- `set_cell_header_format()`
- `set_cell_row_format()`
- `set_border_format()`
- `set_pad_type()`

Tip: The style can also be passed as a keyword argument to `render_table()`

```
self.render_table(
    ['ISBN', 'Title', 'Author'],
    [
        ['99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'],
        ['9971-5-0210-0', 'A Tale of Two Cities', 'Charles Dickens'],
        ['960-425-059-0', 'The Lord of the Rings', 'J. R. R. Tolkien'],
        ['80-902734-1-6', 'And Then There Were None', 'Agatha Christie']
    ]
    style='borderless'
)
```

3.3.1 Spanning Multiple Columns and Rows

To make a table cell that spans multiple columns you can use `table_cell()`, which returns a `TableCell` instance:

```
table = self.table()

table.set_headers(['ISBN', 'Title', 'Author'])
table.set_rows([
    ['99921-58-10-7', 'Divine Comedy', 'Dante Alighieri'],
    self.table_separator(),
    [self.table_cell('This value spans 3 columns.', colspan=3)]
])

table.render()
```

This results in:

```
+---------------+---------------+-----------------+
| ISBN          | Title         | Author          |
+---------------+---------------+-----------------+
| 99921-58-10-7 | Divine Comedy | Dante Alighieri |
+---------------+---------------+-----------------+
| This value spans 3 columns.                     |
+---------------+---------------+-----------------+
```

Tip: You can create a multiple-line page title using a header cell that spans the entire table width:

```
table.set_headers([
    [self.table_cell('Main table title', colspan=3)],
    ['ISBN', 'Title', 'Author']
])
```

This generate:

```
+-------+-------+--------+
| Main table title       |
+-------+-------+--------+
| ISBN  | Title | Author |
+-------+-------+--------+
| ...                    |
+-------+-------+--------+
```

In a similar way you can span multiple rows:

```
table = self.table()

table.set_headers(['ISBN', 'Title', 'Author'])
table.set_rows([
    [
        '978-0521567817',
        'De Monarchia',
        self.table_cell('Dante Alighieri\nspans multiple rows', rowspan=2)
    ]
])

table.render()
```

This outputs:

```
+----------------+---------------+---------------------+
| ISBN           | Title         | Author              |
+----------------+---------------+---------------------+
| 978-0521567817 | De Monarchia  | Dante Alighieri     |
| 978-0804169127 | Divine Comedy | spans multiple rows |
+----------------+---------------+---------------------+
```

You can use the `colspan` and `rowspan` options at the same time which allows you to create any table layout you may wish.

CHAPTER 4

Using Console Commands, Shortcuts and Built-in Commands

In addition to the options you specify for your commands, there are some built-in options as well as a couple of built-in commands for Cleo.

Note: These examples assume you have added a file `application.py` to run at the cli:

```
#!/usr/bin/env python
# application.py

from cleo import Application

application = Application()
# ...

if __name__ == '__main__':
    application.run()
```

4.1 Built-in Commands

The help command lists the help information for the specified command. For example, to get the help for the `list` command:

```
$ python application.py help list
```

Running `help` without specifying a command will list the global options:

```
$ python application.py help
```

4.2 Global Options

You can get help information for any command with the `--help` option. To get help for the `greet` command:

```
$ python application.py greet --help
$ python application.py greet -h
```

You can suppress output with:

```
$ python application.py greet --quiet
$ python application.py greet -q
```

You can get more verbose messages (if this is supported for a command) with:

```
$ python application.py greet --verbose
$ python application.py greet -v
```

If you need more verbose output, use *-vv* or *-vvv*

```
$ python application.py greet -vv
$ python application.py greet -vvv
```

If you set the optional arguments to give your application a name and version:

```
application = Application('console', '1.2')
```

then you can use:

```
$ python application.py --version
$ python application.py -V
```

to get this information output:

```
Console version 1.2
```

If you do not provide both arguments then it will just output:

```
console tool
```

You can force turning on ANSI output coloring with:

```
$ python application.py greet --ansi
```

or turn it off with:

```
$ python application.py greet --no-ansi
```

You can suppress any interactive questions from the command you are running with:

```
$ python application.py greet --no-interaction
$ python application.py greet -n
```

4.3 Shortcut Syntax

You do not have to type out the full command names. You can just type the shortest unambiguous name to run a command. So if there are non-clashing commands, then you can run `help` like this:

```
$ python application.py h
```

CHAPTER 5

Building a Single Command Application

When building a command line tool, you may not need to provide several commands. In such case, having to pass the command name each time is tedious. Fortunately, it is possible to remove this need by using *default()* when adding a command:

```
from cleo import Application

command = GreetCommand()

app = Application()
app.add(command.default())

# this now executes the 'GreetCommand' without passing its name
app.run()
```