chuda Documentation

Release 0.0.21

Romain Moreau

Aug 06, 2019

Contents

Chuda is a very simple Python3 framework to create CLI (Command-Line-Interface) tools.

CHAPTER 1

Features

- Represent commands and argparse arguments by Python classes
- Handle parsing of a configuration file for you (INI, JSON, or YAML with pyyaml)
- Provide you a configurable logger, and some basic options to quiet/verbose mode
- Provide you a simple interface to run some shell commands
- Signals handling by decorator

CHAPTER 2

The User Guide

2.1 Installation of chuda

This part of the documentation covers the installation of chuda. The first step to using any software package is getting it properly installed.

2.1.1 $ pipenv install chuda

To install chuda, simply run this simple command in your terminal of choice:

```
$ pipenv install chuda
```

If you don't have pipenv installed, head over to the Pipenv website for installation instructions. Or, if you prefer to just use pip and don't have it installed, this Python installation guide can guide you through the process.

2.1.2 Get the Source Code

chuda is developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone git://github.com/Varkal/chuda.git
```

Or, download the tarball:

```
$ curl -OL https://github.com/Varkal/chuda/tarball/master
# optionally, zipball is also available (for Windows users).
```

Once you have a copy of the source, you can embed it in your own Python package, or install it into your site-packages easily:

```
$ cd chuda
$ pip install .
```

2.2 Quickstart

Eager to get started? This page gives a good introduction in how to get started with chuda.

First, make sure that:

- chuda is *installed*
- chuda is up-to-date

Let's get started with some simple examples.

2.2.1 Hello World

Create an app with chuda is very simple

```
from chuda import App, autorun

@autorun() # Equivalent to if __name__ == "__main__"
class HelloWorldApp(App):
    def main(self):
        self.logger.info("Hello Word")
```

2.2.2 Parse a config file

Chuda will handle for you the parsing of a configuration file

For example, with this configuration file:

```
# ./config.ini

[hello]
name = John
```

This code will parse it automatically:

```
from chuda import App, autorun

@autorun()
class ConfigApp(App):
    config_path = ["./config.ini", "../config.ini"]

    def main(self):
        self.logger.info("Hello {}".format(self.config["hello"]["name"]))
```

chuda can handle ini (default), json and yaml files (only if pyyaml is installed). You must specified which parser will be used with the *config_parser* attribute

As you can see, *config_path* is a list. Chuda will try to load each file in the order in which they have been declared until it find one that exists.

2.2.3 Handle arguments

Chuda gives you a declarative interface to add options and parameters to your applications

```
from chuda import App, autorun, Option, Parameter

@autorun()
class ArgumentsApp(App):

    arguments = [
        Option(["--language"], default="en", choices=["en", "fr"]),
        Option(["--polite", "-p"], action="store_true"),
        Parameter("name"),
    ]

    def main(self):
        salutation = ""

        if self.arguments.language == "en":
            salutation = "Hello " if self.arguments.polite else "Hi "
        elif self.arguments.language == "fr":
            salutation = "Bonjour " if self.arguments.polite else "Salut "

        salutation += self.arguments.name

        self.logger.info(salutation)
```

An *Option* represent an UNIX style option ("e.g: git commit **-m** stuff")

A *Parameter* represent a simple parameter ("e.g: git checkout **stuff**")

Both takes the same parameters as the `add_argument()` method.

By default, chuda proposes three basic options :

- `-q` / `--quiet` : The logger stop logging
- `-v` / `--verbose` : The logging level is lowered to debug
- `--version` : print the content of the *version* attribute and exit

2.2.4 Create subcommands

Chuda gives you a very simple way to create subcommands in your application by defining *Command* subclasses

```
from chuda import App, autorun, Command

# Should be split in mutiple files

class FirstCommand(Command):
    command_name = "first"
    description = "this is the first command"

    def main(self):
        self.logger.info("first from {}".format(self.app.app_name))


class SecondCommand(Command):
    command_name = "second"
```

(continues on next page)

(continued from previous page)

```
    description = "this is the second command"

    def main(self):
        self.logger.info("second from {}".format(self.app.app_name))


@autorun()
class SubCommandsApp(App):
    app_name = "my_app"

    subcommands = [
        FirstCommand,
        SecondCommand
    ]
```

subcommands is transformed to a dictionnary at runtime. So, if you need to call a subcommand from an other subcommand :

```
from chuda import App, autorun, Command

# Should be split in mutiple files

class FirstCommand(Command):
    command_name = "first"
    description = "this is the first command"

    def main(self):
        self.logger.info("first from {}".format(self.app.app_name))


class SecondCommand(Command):
    command_name = "second"
    description = "this is the second command"

    def main(self):
        self.app.subcommands["first"].run()


@autorun()
class SubCommandsApp(App):
    app_name = "my_app"

    subcommands = [
        FirstCommand,
        SecondCommand
    ]
```

2.2.5 Add plugins

Chuda apps encourage separation of concerns with a system of plugins

A plugin is created by extend the *Plugin* class

```
from chuda import App, autorun, Plugin
import requests
```

(continues on next page)

(continued from previous page)

```
class HttpPlugin(Plugin):
    base_url = None

    def on_create(self):
        self.enrich_app("http", self)

    def on_config_read(self):
        try:
            self.base_url = self.app.config["base_url"]
        except KeyError:
            self.base_url = "http://www.example.com"

    def get_root(self):
        return requests.get(self.base_url)


@autorun()
class HttpApp(App):
    http = None

    plugins = [
        HttpPlugin()
    ]

    def main(self):
        response = self.http.get_root()
        self.logger.info(response.text)
```

Plugins provides mutliple methods to execute code at key points in the lifecycle of the application. See *Plugin* documentation for more informations

CHAPTER 3

The Api Guide

3.1 Developer Interface

3.1.1 Application and Commands

class chuda.app.**App**

Base class for create an application in chuda

app_name = ''

Name of the application, show in the help and version strings

arguments = []

List of *Argument* objects. Replace with the argparse.Namespace at runtime

arguments_declaration = []

arguments will be copied here of before it be replaced with namespace

config = {}

The configuration file will be loaded here

config_parser = 'ini'

The parser used to parse the configuration file. Possible values are: ini, json, yaml

config_path = []

Acceptable paths to find the configuration file. Stop searching on the first one exists

description = ''

Description of the command. Print in help

logger = None

Instance of Logger

override_default_arguments = False

Should *arguments* override default provided arguments ?

merge_arguments_in_subcommands = True

Should *arguments* be merged in subcommands instead of being accesibble globally ?

parser = None
Instance of ArgumentParser

plugins = []
List of Plugins

shell = <chuda.shell.Runner object>
Instance of *Runner*

subcommands = []
List of Command

version = '0.0.1'
version of your application. Display withe –version flag

call_plugins(*step*)
For each plugins, check if a "step" method exist on it, and call it

Parameters **step** (*str*) – The method to search and call on each plugin

run()
Run the application

main()
Main method of the application when the program is started without any subcommand selected. If this is not overrided, it will print the help

class chuda.commands.**Command**
A subcommand for multicommands cli tool

command_name = None
Name of the command, use on the command-line (like "add", in "git add")

use_subconfig = False
Should use config_parser and config_path to generate a config for this command

config = {}
The configuration file will be loaded here

config_parser = 'ini'
The parser used to parse the configuration file. Possible values are: ini, json, yaml

config_path = []
Acceptable paths to find the configuration file. Stop searching on the first one exists

app = None
Contains a reference to the *App* instance who contains this Command

merge_parent_arguments = True
Should parent arguments be merge with local arguments ? True by default.

arguments = []
List of *Argument* objects. Replace with the argparse.Namespace at runtime

arguments_declaration = None

arguments will be copied here of before it be replaced with namespace
Warning: This will contains **only** local arguments declarations

logger = None
Instance of Logger

shell = <chuda.shell.Runner object>
Instance of *Runner*

run()
Run the command

setup(*app*)
Setup properties from parent app on the command

3.1.2 Arguments

class chuda.arguments.**Argument**(*name=None*, *action='store'*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=None*, *help=None*, *metavar=None*, *dest=None*)
Abstract parent class for *Option* and *Parameter*.

For attributes who are not documented here, please see `add_argument()` documentation

convert_to_argument()
Convert the Argument object to a tuple use in `add_argument()` calls on the parser

class chuda.arguments.**Option**(*name=None*, *action='store'*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=None*, *help=None*, *metavar=None*, *dest=None*)
Represent an option on the command-line (`mycommand --whatever`)

name
Options can have multiple names, so name **must** be a list or a tuple

Type list

get_default_name()
Return the default generated name to store value on the parser for this option.

eg. An option *['-s', '–use-ssl']* will generate the *use_ssl* name

Returns the default name of the option

Return type str

convert_to_argument()
Convert the Argument object to a tuple use in `add_argument()` calls on the parser

class chuda.arguments.**Parameter**(*name=None*, *action='store'*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=None*, *help=None*, *metavar=None*, *dest=None*)
Represent a parameter on the command-line (`mycommand whatever`)

name
Parameter can have only one name, so it **must** be a string

Type str

convert_to_argument()
Convert the Argument object to a tuple use in `add_argument()` calls on the parser

3.1.3 Plugins

class chuda.plugins.**Plugin**
Class represent a Plugin for a Chuda application A plugin can register hooks for steps in the application lifecycle or enrich the app with new properties

setup(*app*)
Setup the app on the plugin

enrich_app(*name*, *value*)
Add a new property to the app (with setattr)

Parameters

- **name** (*str*) – the name of the new property
- **value** (*any*) – the value of the new property

on_create()
Called at the app creation

on_arguments_parsed()
Called after arguments have been parsed

on_config_read()
Called after configurations files have been read

on_signals_handled()
Called after signals handlers have been setuped

on_logger_created()
Called after logger have been created

on_run()
Called just before run the app

on_end()
Called after all steps and code have been executed

3.1.4 Decorators

chuda.decorators.**autorun**()
Call the run method of the decorated class if the current file is the main file

chuda.decorators.**signal_handler**(*sig*)
Flag a method to be used as a signal handler

Parameters **sig** (*signal*) – The signal, from the signal module

3.1.5 Shell

class chuda.shell.**Runner**(*logger=None*, *cwd=None*)
Factory for *ShellCommand*

cwd
the current working directory

Type str

logger
Instance of Logger

Type Logger

run(*command*, *block=True*, *cwd=None*, *stdin=-1*, *stdout=-1*, *stderr=-1*)
Create an instance of *ShellCommand* and run it

Parameters

- **command** (*str*) – *ShellCommand*
- **block** (*bool*) – See *ShellCommand*
- **cwd** (*str*) – Override the runner cwd. Useb by the *ShellCommand* instance

class chuda.shell.**ShellCommand**(*command*, *logger*, *cwd=None*, *block=True*, *stdin=-1*, *stdout=-1*, *stderr=-1*)

DEPRECATED: Please use sh.py instead

Abstraction layer for shell subprocess

You can disable stdout, stdin, stderr

block
if false, the command will be run asynchronously

Type bool

command
the shell command to run

Type str

cwd
the current working directory

Type str

error
everything the command will write on stderr will be here

Type str|list

logger
Instance of Logger

Type Logger

output
everything the command will write on stdout will be here

Type str|list

process
the Popen instance use to run the command

Type Popen

return_code
the return code of the shell command

Type int

thread
the Thread instance use if the command is run in non blocking mode

Type Thread

writer
Instance of TextIOWrapper plugged on stdin

Type TextIOWrapper

run()
Run the shell command

Returns return this ShellCommand instance for chaining

Return type *ShellCommand*

send(*value*)
Send text to stdin. Can only be used on non blocking commands

Parameters **value** (*str*) – the text to write on stdin

Raises TypeError – If command is blocking

Returns return this ShellCommand instance for chaining

Return type *ShellCommand*

poll_output()
Append lines from stdout to self.output.

Returns The lines added since last call

Return type list

poll_error()
Append lines from stderr to self.errors.

Returns The lines added since last call

Return type list

kill()
Kill the current non blocking command

Raises TypeError – If command is blocking

wait_for(*pattern*, *timeout=None*)
Block until a pattern have been found in stdout and stderr

Parameters

- **pattern** (Pattern) – The pattern to search
- **timeout** (*int*) – Maximum number of second to wait. If None, wait infinitely

Raises TimeoutError – When timeout is reach

is_running()
Check if the command is currently running

Returns True if running, else False

Return type bool

wait()
Block until the end of the process

print_live_output()
Block and print the output of the command

Raises TypeError – If command is blocking

CHAPTER 4

Miscellaneous

4.1 Release and Version History

4.1.1 0.0.21 (2018-06-13)

First “public” version

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

P

R

S

T

U

V

W