Clinner Documentation

Release 1.9.4

José Antonio Perdiguero López

Jul 18, 2018

Contents:

CHAPTER 1

Installation

1.1 Quick Start

1. Install this package using pip:

```
pip install clinner
```

2. Create a command

```
from clinner.command import command

@command
def foo(*args, **kwargs):
    return True
```

3. Create a main file:

```
from clinner.run.main import Main

if __name__ == '__main__':
    sys.exit(Main().run())
```

1.2 Settings

Clinner settings can be specified through **CLINNER_SETTINGS** environment variable or using `-s` or `--settings` command line flags during invocation. The format to specify settings module or class should be either `package.module` or `package.module:Class`.

1.2.1 Default Arguments

Default arguments for commands. Let a command `foo` declared:

```
default_args = {
    'foo': ['-v', '--bar', 'foobar'],
}
```

CHAPTER 2

Clinner

2.1 Commands

2.1.1 Command Decorator

Commands are declared using a decorator to register given functions. Commands are functions with the follow parameters:

func Function that will be called when command would be executed.

command_type Type of the command, could be a *bash* or *python* command.

args Parser arguments for this command.

parser_opts Command subparser's keywords, such as description.

class command(*func=None*, *command_type=<Type.PYTHON: 'python'>*, *args=None*, *parser_opts=None*)

Decorator to register the given functions in a register, along with their command line arguments.

How to use

This decorator allows to be used as a common decorator without arguments, where default type (*python*) will be used:

```
@command
def foobar(bar):
    pass
```

Or specifying the type:

```
@command(command_type=Type.SHELL)
def foobar(bar):
    return [['cat', 'foobar']]
```

But also is possible to provide command line arguments, as expected by `argparse.ArgumentParser.add_argument()`:

```
@command(args=((('-f', '--foo'), {'help': 'Foo argument that does nothing'}),
               (('--bar',), {'action': 'store_true', 'help': 'Bar argument stored as␣
↪True'})),
         parser_opts={'title': 'foobar_command', 'help': 'Help for foobar_command'})
def foobar(*args, **kwargs):
    pass
```

For last, is possible to decorate functions or class methods:

```
class Foo:
    @staticmethod
    @command
    def bar():
        pass
```

Types

Define the type of process to be executed by `clinner.run.Main`.

Python

Python function executed in a different process. Must implement the function itself.

Shell

List of shell commands executed in different processes. Each command must be a list of splitted command such as returned from `shlex.split()`. As it can execute more than a single command, a list of lists should be returned.

Bash

Alias for Shell.

Arguments

Command line arguments are defined through *args* parameter of command decorator. This arguments can be defined using the follow structure:

```
@command(args=(
    (('positionals',) {'help': 'Positional arguments', 'nargs': '+'}),
    (('-f', '--foo'), {'help': 'Foo argument', 'default': 'foo'}),
    (('--bar',), {'help': 'Bar argument', 'default': 1, 'type': int, 'choices':␣
↪range(1, 6)}),
))
def cmd(*args, **kwargs):
    pass
```

Also is possible to define args using a callable that receives the parser:

```
def add_arguments(parser):
    parser.add_argument('positionals', help='Positional arguments', nargs='+')
    parser.add_argument('-f', '--foo', help='Foo argument', default='foo')
    parser.add_argument('--bar', help='Bar argument', default=1, type=int,␣
↪choices=range(1, 6))

@command(args=add_arguments)
def cmd(*args, **kwargs):
    pass
```

Parser options

It is possible to pass options to the command parser, such as *title*, *help*... These options should be passed through *parser_opts* parameter of command decorator:

```
@command(parser_opts={'help': 'Command doing awesome things!'})
def cmd(*args, **kwargs):
    pass
```

2.1.2 Register

All commands will be registered in a *clinner.command.CommandRegister* that can be accessed through `command.register`. Each entry in this register is a dictionary with the fields declared at the beginning of this section.

class CommandRegister
Register for commands.

2.2 Main

A main class is defined to ease the creation of command line applications. This class follows the process:

1. Create a parser using `argparse.ArgumentParser` for the application:
   (a) Calling all `add_arguments(parser)` methods from all super classes, e.g: `clinner.mixins.HealthCheckMixin`.
   (b) Addding a subparser for each command with their specific arguments.
2. Parse arguments using the argument parser created previously.
3. Inject variables into environment calling all super classes methods whose name starts with `inject_`.
4. Load settings module from **CLINNER_SETTINGS** environment variable. More details below.

class Main(*args=None*, *parse_args=True*)

add_arguments(*parser: argparse.ArgumentParser*)
Add to parser all necessary arguments for this Main.

Parameters **parser** – Argument parser.

inject()
Add all environment variables defined in all inject methods.

run(**args*, *command=None*, ***kwargs*)
Run specified command through system arguments.

Arguments that have been parsed properly will be passed through **kwargs. Unknown arguments will be passed as a list of strings through *args.

This method will print a header and the return code.

Parameters **command** – Explicit command. Use that command instead of the one passed by shell arguments.

2.2.1 Commands

All commands previously loaded will be available to use by the main class but also there is a another mechanism to load commands using the main class. To do this simply specify a list of fully qualified name commands, e.g: Given a module `foo` with a command `bar`:

```
from clinner.run.main import Main


class FooMain(Main):
    commands = (
        'foo.bar',
    )
```

This bar command will be assigned as a `staticmethod` to FooMain class to provide an easy access: `FooMain.bar()`.

In case of overriding a command already imported, the one defined in the class will prevail, e.g:

```
from clinner.run.main import Main


class FooMain(Main):
    commands = (
        'foo.bar',
    )

    @staticmethod
    @command
    def bar(*args, **kwargs):
        pass  # This command will be the executed instead of foo.bar
```

2.2.2 Mixins

Clinner provides some useful mixins for main classes that adds different behaviors to these classes.

class HealthCheckMixin
Adds health checking behavior to Main classes. To do that is necessary to define a health_check method responsible of return the current status of the application.

This mixin also adds a new parameter `-r`, `--retry` that defines the number of retries done after a failure. These retries uses an exponential backoff to calculate timing.

health_check()
Does a health check.

Returns True if health check was successful. False otherwise.

run(**args*, ***kwargs*)
Run specified command through system arguments.

Before running the command, a health check function will be called and if result is not successful, the command will be aborted.

Arguments that have been parsed properly will be passed through **kwargs. Unknown arguments will be passed as a list of strings through *args.

This method will print a header and the return code.

2.3 Django Commands

Using previously defined Main classes it's possible to wrap it as a Django command:

```
from clinner.run.main import Main


class FooMain(Main):
    description = 'Foo main'

    commands = (
        'foo.bar',
    )

class FooDjangoCommand(DjangoCommand):
    main_class = FooMain
```

This class handles the django commands arguments as well as passing them to run method.

class DjangoCommand(**args*, ***kwargs*)
Wrapper that makes a Django command from a Main class, including parsers only for commands.

2.4 Examples

Some Clinner examples.

2.4.1 Simple Main

Example of a simple main with two defined commands *foo* and *bar*.

```
#!/usr/bin/env python
import os
import shlex
import sys

from clinner.command import command, Type as CommandType
from clinner.run.main import Main


@command(command_type=CommandType.SHELL
         args=(('-i', '--input'),
               ('-o', '--output')),
```

(continues on next page)

(continued from previous page)

```
        parser_opts={'help': 'Foo command'})
def foo(*args, **kwargs):
    """List of foo commands"""
    ls_cmd = shlex.split('ls')
    wc_cmd = shlex.split('wc')
    wc_cmd += [kwargs['input'], kwargs['output']]

    return [ls_cmd, wc_cmd]


@command(command_type=CommandType.PYTHON,
        parser_opts={'help': 'Bar command'})
def bar(*args, **kwargs):
    """Do a bar."""
    return True


if __name__ == '__main__':
    sys.exit(Main().run())
```

2.4.2 Builder Main

Example of main module with build utilities such as unit tests, lint, sphinx doc, tox and dist packaging:

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

from clinner.run import Main


class Build(Main):
    commands = (
        'clinner.run.commands.pytest.pytest',
        'clinner.run.commands.prospector.prospector',
        'clinner.run.commands.sphinx.sphinx',
        'clinner.run.commands.tox.tox',
        'clinner.run.commands.dist.dist',
    )


if __name__ == '__main__':
    sys.exit(Build().run())
```

2.4.3 Django Main

Example of main module for a Django application that uses uwsgi, health-check, prospector, pytest.

```
#!/usr/bin/env python3.6
"""Run script.
"""
import argparse
import multiprocessing
import os
```

(continues on next page)

(continued from previous page)

```
import shlex
import sys
from socket import gethostname
from typing import List

import hvac
from clinner.command import Type as CommandType, command
from clinner.run import HealthCheckMixin, Main as BaseMain
from django.core.exceptions import ImproperlyConfigured

PYTHON = 'python3.6'
COVERAGE = 'coverage'
PROSPECTOR = 'prospector'
HEALTH_CHECK = 'health_check'

BASE_DIR = os.path.dirname(os.path.abspath(__file__))


@command(command_type=CommandType.SHELL)
def migrate(*args, **kwargs) -> List[List[str]]:
    cmd = shlex.split(f'{PYTHON} manage.py migrate')
    cmd += args
    return [cmd]


@command(command_type=CommandType.SHELL)
def build(*args, **kwargs) -> List[List[str]]:
    return migrate('--fake-initial') + collectstatic('--noinput')


@command(command_type=CommandType.SHELL)
def manage(*args, **kwargs) -> List[List[str]]:
    cmd = shlex.split(f'{PYTHON} manage.py')
    cmd += args
    return [cmd]


@command(command_type=CommandType.SHELL)
def unit_tests(*args, **kwargs) -> List[List[str]]:
    parallel_count = multiprocessing.cpu_count()
    coverage_erase = shlex.split(f'{COVERAGE} erase')

    tests = shlex.split(f'{COVERAGE} run --concurrency=multiprocessing manage.py test␣
↪--parallel {parallel_count}')
    tests += args

    coverage_combine = shlex.split(f'{COVERAGE} combine')
    coverage_report = shlex.split(f'{COVERAGE} report')
    coverage_xml = shlex.split(f'{COVERAGE} xml')
    coverage_html = shlex.split(f'{COVERAGE} html')

    return [coverage_erase, tests, coverage_combine, coverage_xml, coverage_html,␣
↪coverage_report]


@command(command_type=CommandType.SHELL)
def prospector(*args, **kwargs) -> List[List[str]]:
```

(continues on next page)

(continued from previous page)

```
    cmd = [PROSPECTOR]
    cmd += args
    return [cmd]


@command(command_type=CommandType.SHELL)
def runserver(*args, **kwargs) -> List[List[str]]:
    cmd = shlex.split(f'{PYTHON} manage.py runserver --nothreading')
    cmd += args
    return migrate('--fake-initial') + [cmd]


@command(command_type=CommandType.SHELL)
def uwsgi(*args, **kwargs) -> List[List[str]]:
    http = f':{os.environ["APP_PORT"]}'
    stats = f':{os.environ["STATS_PORT"]}'
    ini = 'uwsgi.ini'
    cmd = ['uwsgi', '--http', http, '--stats', stats, '--ini', ini]
    cmd += args
    return migrate('--fake-initial') + [cmd]


@command(command_type=CommandType.SHELL)
def collectstatic(*args, **kwargs) -> List[List[str]]:
    cmd = shlex.split(f'{PYTHON} manage.py collectstatic')
    cmd += args
    return [cmd]


@command(command_type=CommandType.SHELL)
def shell(*args, **kwargs) -> List[List[str]]:
    cmd = shlex.split(f'{PYTHON} manage.py shell')
    cmd += args
    return [cmd]


@command(command_type=CommandType.SHELL)
def health_check(*args, **kwargs) -> List[List[str]]:
    """
    Run health-check
    """
    cmd = [HEALTH_CHECK]
    cmd += args
    return [cmd]


class Main(HealthCheckMixin, BaseMain):
    commands = (
        'migrate',
        'build',
        'manage',
        'unit_tests',
        'prospector',
        'runserver',
        'uwsgi',
        'collectstatic',
        'shell',
```

(continues on next page)

(continued from previous page)

```
        'health_check',
    )

    def add_arguments(self, parser: argparse.ArgumentParser):
        parser.add_argument('-s', '--settings', default='Development', help='Settings␣
↪module')

    def inject_app_settings(self):
        """
        Injecting own settings.
        """
        config_name = self.args.settings
        os.environ['APP_HOST'] = os.environ.get('HOSTNAME', os.environ.get('APP_HOST',
↪ '0.0.0.0'))
        os.environ['APP_PORT'] = os.environ.get('PORT_8000', os.environ.get('APP_PORT
↪', '8000'))
        os.environ['STATS_PORT'] = os.environ.get('PORT_9000', os.environ.get('STATS_
↪PORT', '9000'))

        # Django
        os.environ['DJANGO_SETTINGS_MODULE'] = 'your_app.settings'
        os.environ['DJANGO_CONFIGURATION'] = self.args.settings

        # Plugins
        os.environ['CLINNER_SETTINGS'] = self.args.settings or f'your_app.plugins_
↪settings.clinner:{self.args.settings}'
        os.environ['HEALTH_CHECK_SETTINGS'] = f'your_app.plugins_settings.health_
↪check:{self.args.settings}'

    def health_check(self):
        """
        Does a check using Health Check application.

        :return: 0 if healthy.
            """
            return not self.run_command('manage', 'health_check', 'health', '-e')


    if __name__ == '__main__':
        sys.exit(Main().run())
```

CHAPTER 3

Batteries Included

3.1 Inputs

Clinner provides some useful functions to ask for a user input.

bool_input(*input_str: str*) → str
Prints a message asking for a yes/no response, otherwise it will continue asking.

Parameters **input_str** – Message to print.

Returns User response.

choices_input(*input_str: str, choices: List[Any]*) → str
Prints a message asking for a choice of given values.

Parameters

- **input_str** – Message to print.
- **choices** – Choices.

Returns User response.

default_input(*input_str: str*, *default: Any = None*) → str
Prints a message offering a default value.

Parameters

- **input_str** – Message to print.
- **default** – Default value.

Returns User response.

3.2 Commands

Clinner provides some defined commands ready to be used by Main classes.

dist
Bump version, create package and upload it.

nose
Run unit tests using Nose.

pytest
Run unit tests using pytest.

prospector
Run prospector lint.

sphinx
Run an sphinx command.

tox
Run tests using tox.

3.3 Mixins

Clinner provides some useful mixins for main classes that adds different behaviors to these classes.

class HealthCheckMixin
Adds health checking behavior to Main classes. To do that is necessary to define a health_check method responsible of return the current status of the application.

This mixin also adds a new parameter `-r`, `--retry` that defines the number of retries done after a failure. These retries uses an exponential backoff to calculate timing.

health_check()
Does a health check.

Returns True if health check was successful. False otherwise.

run(**args*, ***kwargs*)
Run specified command through system arguments.

Before running the command, a health check function will be called and if result is not successful, the command will be aborted.

Arguments that have been parsed properly will be passed through **kwargs. Unknown arguments will be passed as a list of strings through *args.

This method will print a header and the return code.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index