Cfg4Py Documentation

Release 0.9.3

Aaron Yang

Jun 03, 2022

CONTENTS:

1 Overview **1**
- 1.1 Features . . . 1

2 Installation **5**
- 2.1 Stable release . . . 5
- 2.2 From sources . . . 5

3 Usage **7**
- 3.1 Quick Guide . . . 7
- 3.2 Exhausted Guide . . . 7
- 3.3 Use cfg4py as a cheat sheet . . . 9

4 cfg4py **11**
- 4.1 cfg4py package . . . 11

5 Contributing **15**
- 5.1 Types of Contributions . . . 15
- 5.2 Get Started! . . . 16
- 5.3 Pull Request Guidelines . . . 17
- 5.4 Tips . . . 17
- 5.5 Deploying . . . 17

6 Credits **19**
- 6.1 Development Lead . . . 19
- 6.2 Contributors . . . 19

7 History **21**
- 7.1 0.1.0 (2020-04-06) . . . 21
- 7.2 0.5.0 (2020-04-30) . . . 21
- 7.3 0.7.0 (2020-10-31) . . . 21
- 7.4 0.8.0 (2020-11-22) . . . 21
- 7.5 0.9.0 (2020-12-03) . . . 21
- 7.6 0.9.2 (2021-12-17) . . . 22
- 7.7 0.9.3 (2022-06-03) . . . 22

8 FAQ **23**

9 Indices and tables **25**

Python Module Index **27**

Index 29

CHAPTER

ONE

OVERVIEW

- Free software: BSD license
- Documentation: https://cfg4py.readthedocs.io.

A python config module that:

1. Adaptive deployment (default, dev, test, production) support
2. Cascading configuration (central vs local) support
3. Auto-complete
4. Templates (logging, database, cache, message queue,. . .)
5. Environment variables macro support
6. Enable logging in one line
7. Built on top of yaml

1.1 Features

It's common to see that you have different settings for development machine, test machine and production site. They share many common settings, but a few of them has to be different.

For example, developers should connect to local database server when performing unittest, and tester should connect to their own database server. All these servers should be deployed separately and no data should be messed up.

Cfg4Py has perfect solution supporting for this: adaptive deployment environment support.

1.1.1 Adaptive Deployment Environment Support

In any serious projects, your application may run at both development, testing and production site. Except for effort of copying similar settings here and there, sometimes we'll mess up with development environment and production site. Once this happen, it could result in very serious consequence.

To solve this, Cfg4Py developed a mechanism, that you provide different sets for configurations: dev for development machine, test for testing environment and production for production site, and all common settings are put into a file called *defaults*.

cfg4py module knows which environment it's running on by looking up environment variable __cfg4py_server_role__. It should be one of *DEV*, *TEST* and *PRODUCTION*. If nothing found, it means setup is not finished, and Cfg4Py will refuse to work. If the environment is set, then Cfg4Py will read settings from defaults set, then apply update from either of *DEV*, *TEST* and *PRODUCTION* set, according to the environment the application is running on.

Important: Since 0.9.0, cfg4py can still work if __cfg4py_server_role__ is not set, when it work at non-strict mode.

1.1.2 Cascading design

Assuming you have a bunch of severs for load-balance, which usually share same configurations. So you'd like put the configurations on a central repository, which could be a redis server or a relational database. Once you update configuration settings at central repository, you update configurations for all servers. But somehow for troubleshooting or maintenance purpose, you'd like some machines could have its own settings at a particular moment.

This is how Cfg4Py solves the problem:

1. Configure your application general settings at remote service, then implement a *RemoteConfigFetcher* (Cfg4Py has already implemented one, that read settings from redis), which pull configuration from remote serivce periodically.
2. Change the settings on local machine, after the period you've set, these changes are popluated to all machines.

1.1.3 Auto-complete

With other python config module, you have to remember all the configuration keys, and refer to each settings by something like cfg[“services”][“redis”][“host”] and etc. Keys are hard to rememb, prone to typo, and way too much tedious.

When cfg4py load raw settigns from yaml file, it'll compile all the settings into a Python class, then Cfg4Py let you access your settings by attributes. Compares the two ways to access configure item:

```
cfg["services"]["redis"]["host"]
```

vs:

```
cfg.services.redis.host
```

Apparently the latter is the better.

And, if you trigger a build against your configurations, it'll generate a python class file. After you import this file (named 'schema.py') into your project, then you can enjoy code auto-complete!

1.1.4 Templates

It's hard to remember how to configure log, database, cache and etc, so cfg4py provide templates.

Just run cfg4py scaffold, follow the tips then you're done.

```
(lynch) aaron@DESKTOP-UUP754F:~/miniconda3$ cfg4py scaffold
Creating a configuration boilerplate:
Where should I save configuration files?
/workspace/tmp
Which flavors do you want?
--------------------

    0  - console + rotating file logging
    10 - redis/redis-py (gh://andymccurdy/redis-py)
    11 - redis/aioredis (gh://aio-libs/aioredis)
    20 - mysql/PyMySQL (gh://PyMySQL/PyMySQL)
    30 - postgres/asyncpg (gh://MagicStack/asyncpg)
    31 - postgres/psycopg2 (gh://psycopg/psycopg2)
    40 - mq/pika (gh://pika/pika)
    50 - mongodb/pymongo (gh://mongodb/mongo-python-driver)

Please choose flavors by index, separated each by a comma(,):
```

1.1.5 Environment variables macro

The best way to keep secret, is never share them. If you put account/password files, and these files may be leak to the public. For example, push to github by accident.

With cfg4py, you can set these secret as environment variables, then use marco in config files. For example, if you have the following in defaults.yaml (any other files will do too):

```
postgres:
        dsn: postgres://${postgres_account}:${postgres_password}@localhost
```

then cfg4py will lookup postgres_account, postgres_password from environment variables and make replacement.

1.1.6 Enable logging with one line

with one line, you can enable file-rotating logging:

```
cfg.enable_logging(level, filename=None)
```

1.1.7 Apply configuration change on-the-fly

Cfg4Py provides mechanism to automatically apply configuration changes without restart your application. For local files configuration change, it may take effect immediately. For remote config change, it take effect up to *refresh_interval* settings.

1.1.8 On top of yaml

The raw config format is backed by yaml, with macro enhancement. YAML is the best for configurations.

1.1.9 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Cfg4Py, run this command in your terminal:

```
$ pip install cfg4py
```

This is the preferred method to install Cfg4Py, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Cfg4Py can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone
```

Or download the tarball:

```
$ curl -OJL https://github.com/zillionare/cfg4py/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

3.1 Quick Guide

To use Cfg4Py in a project:

```
import cfg4py

# create config object
cfg = cfg4py.init(path_to_config_dir)

# then refer to settings by cfg's properties
# given the following yaml settings (filename: defaults.yaml) under path_to_config_dir

# services:
#   redis:
#       host: localhost

# you can access settings by '.'
print(cfg.services.redis.host)

# you CANNOT access settings like this way (this will raise exceptions):
print(cfg["services"])
```

3.2 Exhausted Guide

3.2.1 Step 1.

Use Cfg4Py tool to generate configuration templates:

```
cfg4py scaffold
```

The output is:

```
(lynch) aaron@DESKTOP-UUP754F:~/miniconda3$ cfg4py scaffold
Creating a configuration boilerplate:
Where should I save configuration files?
/workspace/tmp
Which flavors do you want?
--------------------

    0  - console + rotating file logging
    10 - redis/redis-py (gh://andymccurdy/redis-py)
    11 - redis/aioredis (gh://aio-libs/aioredis)
    20 - mysql/PyMySQL (gh://PyMySQL/PyMySQL)
    30 - postgres/asyncpg (gh://MagicStack/asyncpg)
    31 - postgres/psycopg2 (gh://psycopg/psycopg2)
    40 - mq/pika (gh://pika/pika)
    50 - mongodb/pymongo (gh://mongodb/mongo-python-driver)

Please choose flavors by index, separated each by a comma(,):
```

You may need modify settings according to your enviroment.

3.2.2 Step 2.

Build config class, and import it into your project:

```
cfg4py build /path/to/your/config/dir
```

```
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    # make sure that schema is at your $PYTHONPATH
    from schema import Config
import cfg4py

cfg: Config = cfg4py.init('/path/to/your/config/dir')

# now you should be able to get auto-complete hint while typing
cfg.?
```

3.2.3 Step 3.

cfg4py will take care of setting's change automatically, all you need to do is put correct settings into one of (defaults, dev, test, production) config file. And once you change the settings, it should take effect immediately.

To enable cascading config, you can configure a remote source by implemented a subclass of *RemoteConfigFetcher*. A redis fetcher is provided out-of-box:

```
from cfg4py import RedisConfigFetcher
from redis import StrictRedis

cfg = cfg4py.int()  # since we're using remote config now, so we can omit path param
↪here
fetcher = RedisConfigFetcher(key="my_app_config")
logger.info("configuring a remote fetcher")
cfg4py.config_remote_fetcher(fetcher, 1)
```

The settings in redis under *key* should be a json string, which can be converted into a dict object.

3.2.4 Step 4.

Before starting run your application, you should set __cfg4py_server_role__ to any of [DEV,TEST,PRODUCTION] (since 0.9.0, required only if you specified as *strict* mode). You can run the following command to get the help:

```
cfg4py hint set_server_role
```

Hint: since 0.9.0, you can skip this step, if you don't need adaptive deployment support.

3.3 Use cfg4py as a cheat sheet

cfg4py does more than a config module, it can be a cheat sheet for many configurations. For example, want to change pip source (usually you'll if you're in china mainland):

```
cfg4py hint pip

>
- tsinghua: pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
- aliyun: pip config set global.index-url https://mirrors.aliyun.com/pypi/simple/
- tencent: pip config set global.index-url http://mirrors.cloud.tencent.com/pypi/
↪simple
- douban: pip config set global.index-url http://pypi.douban.com/simple/
```

for more, explore by yourself by typing *cfg4py hint*

CHAPTER

FOUR

CFG4PY

4.1 cfg4py package

4.1.1 Submodules

4.1.2 cfg4py.command_line module

class cfg4py.cli.**Command**

Bases: object

build(*config_dir: str*)

Compile configuration files into python script, which is used by IDE's auto-complete function

Args: config_dir: The folder where your configuration files located

Returns:

hint(*what: str = None*, *usage: bool = False*)

show a cheat sheet for configurations. for example:

cfg4py hint mysql

this will print how to configure PyMySQL :param what :param usage

scaffold(*dst: Optional[str]*)

Creates initial configuration files based on our choices. Args:

dst:

Returns:

set_server_role()

version()

cfg4py.cli.**main**()

4.1.3 cfg4py.config module

class cfg4py.config.**Config**
Bases: object

4.1.4 cfg4py.core module

Main module.

class cfg4py.core.**LocalConfigChangeHandler**
Bases: watchdog.events.FileSystemEventHandler

dispatch(*event*)
Dispatches events to the appropriate methods.

Parameters **event** (FileSystemEvent) – The event object representing the file system event.

class cfg4py.core.**RedisConfigFetcher**(*key: str*, *host: str = 'localhost'*, *port: int = 6379*, *db: int = 0*, ***kwargs*)
Bases: *cfg4py.core.RemoteConfigFetcher*

fetch() → dict

class cfg4py.core.**RemoteConfigFetcher**
Bases: object

fetch() → str

cfg4py.core.**build**(*save_to: str*)

cfg4py.core.**config_remote_fetcher**(*fetcher:* cfg4py.core.RemoteConfigFetcher, *interval: int = 300*)

config a remote configuration fetcher, which will pull the settings on every *refresh_interval*

Args: fetcher: sub class of *RemoteConfigFetcher* interval: how long should cfg4py to pull the configuration from remote

Returns:

cfg4py.core.**config_server_role**(*role: str*)

cfg4py.core.**enable_logging**(*level=20*, *log_file=None*, *file_size=10*, *file_count=7*)
Enable basic log function for the application

if log_file is None, then it'll provide console logging, otherwise, the console logging is turned off, all events will be logged into the provided file.

Args: level: the log level, one of logging.DEBUG, logging.INFO, logging.WARNING, logging.Error log_file: the absolute file path for the log. file_size: file size in MB unit file_count: how many backup files leaved in disk

Returns: None

cfg4py.core.**get_config_dir**()

cfg4py.core.**init**(*local_cfg_path: str = None*, *dump_on_change=True*, *strict=False*)
create cfg object. Args:

local_cfg_path: the directory name where your configuration files exist dump_on_change: if configuration is updated, whether or not to dump them into

log file

Returns:

cfg4py.core.**update_config**(*conf: dict*)

cfg4py.core.**yaml_dump**(*conf*, *options=None*)

4.1.5 Module contents

Top-level package for Cfg4Py.

class cfg4py.**RemoteConfigFetcher**

Bases: object

fetch() → str

cfg4py.**enable_logging**(*level=20*, *log_file=None*, *file_size=10*, *file_count=7*)

Enable basic log function for the application

if log_file is None, then it'll provide console logging, otherwise, the console logging is turned off, all events will be logged into the provided file.

Args: level: the log level, one of logging.DEBUG, logging.INFO, logging.WARNING, logging.Error log_file: the absolute file path for the log. file_size: file size in MB unit file_count: how many backup files leaved in disk

Returns: None

cfg4py.**config_remote_fetcher**(*fetcher:* cfg4py.core.RemoteConfigFetcher, *interval: int = 300*)

config a remote configuration fetcher, which will pull the settings on every *refresh_interval*

Args: fetcher: sub class of *RemoteConfigFetcher* interval: how long should cfg4py to pull the configuration from remote

Returns:

cfg4py.**init**(*local_cfg_path: str = None*, *dump_on_change=True*, *strict=False*)

create cfg object. Args:

local_cfg_path: the directory name where your configuration files exist dump_on_change: if configuration is updated, whether or not to dump them into

log file

Returns:

cfg4py.**update_config**(*conf: dict*)

class cfg4py.**RedisConfigFetcher**(*key: str*, *host: str = 'localhost'*, *port: int = 6379*, *db: int = 0*, ***kwargs*)

Bases: *cfg4py.core.RemoteConfigFetcher*

fetch() → dict

cfg4py.**config_server_role**(*role: str*)

cfg4py.**get_instance**()

cfg4py.**get_config_dir**()

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/jieyu_tech/cfg4py/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Cfg4Py could always use more documentation, whether as part of the official Cfg4Py docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jieyu_tech/cfg4py/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *cfg4py* for local development.

1. Fork the *cfg4py* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/cfg4py.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cfg4py
$ cd cfg4py/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 cfg4py tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.com/jieyu_tech/cfg4py/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_cfg4py
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SIX

CREDITS

6.1 Development Lead

- Aaron Yang <aaron_yang@jieyu.ai>

6.2 Contributors

None yet. Why not be the first?

CHAPTER

SEVEN

HISTORY

7.1 0.1.0 (2020-04-06)

- First release on PyPI.

7.2 0.5.0 (2020-04-30)

- add command hint, set_server_role
- export envar
- add pip source, conda source

7.3 0.7.0 (2020-10-31)

- support environment macro

7.4 0.8.0 (2020-11-22)

- rename cfg4py_auto_gen.py to schema.py

7.5 0.9.0 (2020-12-03)

- add strict mode: default is non-strict mode, which allows you run cfg4py without set environment variable __cfg4py_server_role__

  this is a **break** change. If you've used cfg4py in your project and it worked well, after upgrade to 0.9.0, you have to modify your init code as this:

```
cfg4py.init('path_to_config_dir', strict = True)

see more in usage and FAQ document
```

7.6 0.9.2 (2021-12-17)

- hot-reload will now only react to configuration files's change. Check issue 1 here.

7.7 0.9.3 (2022-06-03)

- on apple m1, it's not able to watch file changes, and cause cfg4py fail. This revision will disable hot-reload in such scenario and user can still use all other functions of cfg4py.
- remove support for python 3.6 since it's out of service, and opt 3.10, 3.11 in
- log settings are now available by *cfg.logging*. Check issue 4 here

CHAPTER

EIGHT

FAQ

1. What is schema.py?

   It's generated for code completion. It's safe to keep it in both development environment and release package. Don't try to instantiate it (an TypeError will raise to prevent from instantiate it), you should only use it for typing annotation.

2. Why after upgrade to 0.9.0, cfg4py doesn't work as before?

   v0.9 introduced *strict* mode, which is False by default. When cfg4py is initialized with `strict = True`, cfg4py works only if __cfg4py_server_role__ is set; if it's non-strict mode, cfg4py works with __cfg4py_server_role__ is not set.

   So if you've used cfg4py for a while and it worked before v0.9, then you need to modify your code where it initialize cfg4py as:

   ```
   import cfg4py

   # strict is an added param
   cfg4py.init('path_to_config_dir_as_before', strict = True)
   ```

   if you don't specify *strict = True*, cfg4py still works, but it will NOT read config under the name 'dev.yaml', 'test.yaml' or 'production.yaml'

3. Why *cfg.logging* acts like dict?

   Because *cfg.logging*` is a dict. *cfg.logging* is provided since 0.9.3, in case one may need it, for example, get the log file location. However, logging settings may contains key that is python's reserved word, thus it's not possible to convert it into python's object (It's not allowed to use python's reserved word as object's member)

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

cfg4py, 13
cfg4py.cli, 11
cfg4py.config, 12
cfg4py.core, 12

INDEX

B

build() (*cfg4py.cli.Command method*), 11
build() (*in module cfg4py.core*), 12

C

cfg4py
    module, 13
cfg4py.cli
    module, 11
cfg4py.config
    module, 12
cfg4py.core
    module, 12
Command (*class in cfg4py.cli*), 11
Config (*class in cfg4py.config*), 12
config_remote_fetcher() (*in module cfg4py*), 13
config_remote_fetcher() (*in module cfg4py.core*), 12
config_server_role() (*in module cfg4py*), 13
config_server_role() (*in module cfg4py.core*), 12

D

dispatch() (*cfg4py.core.LocalConfigChangeHandler method*), 12

E

enable_logging() (*in module cfg4py*), 13
enable_logging() (*in module cfg4py.core*), 12

F

fetch() (*cfg4py.core.RedisConfigFetcher method*), 12
fetch() (*cfg4py.core.RemoteConfigFetcher method*), 12
fetch() (*cfg4py.RedisConfigFetcher method*), 13
fetch() (*cfg4py.RemoteConfigFetcher method*), 13

G

get_config_dir() (*in module cfg4py*), 13
get_config_dir() (*in module cfg4py.core*), 12
get_instance() (*in module cfg4py*), 13

H

hint() (*cfg4py.cli.Command method*), 11

I

init() (*in module cfg4py*), 13
init() (*in module cfg4py.core*), 12

L

LocalConfigChangeHandler (*class in cfg4py.core*), 12

M

main() (*in module cfg4py.cli*), 11
module
    cfg4py, 13
    cfg4py.cli, 11
    cfg4py.config, 12
    cfg4py.core, 12

R

RedisConfigFetcher (*class in cfg4py*), 13
RedisConfigFetcher (*class in cfg4py.core*), 12
RemoteConfigFetcher (*class in cfg4py*), 13
RemoteConfigFetcher (*class in cfg4py.core*), 12

S

scaffold() (*cfg4py.cli.Command method*), 11
set_server_role() (*cfg4py.cli.Command method*), 11

U

update_config() (*in module cfg4py*), 13
update_config() (*in module cfg4py.core*), 13

V

version() (*cfg4py.cli.Command method*), 11

Y

yaml_dump() (*in module cfg4py.core*), 13