Piny

Release 1.1.0

Vitaly Samigullin

Sep 22, 2023

USER DOCUMENTATION

Piny is YAML config loader with environment variables interpolation for Python.

- Keep your app's configuration in a YAML file.
- Mark up sensitive data in config as *environment variables*.
- Set environment variables on application deployment.
- Let *Piny* load your configuration file and substitute environment variables with their values.

Piny is developed with Docker and Kubernetes in mind, though it's not limited to any deployment system.

CHAPTER

ONE

SIMPLE EXAMPLE

Set your environment variables, mark up your configuration file with them:

```yaml
db:
  login: user
  password: ${DB_PASSWORD}
mail:
  login: user
  password: ${MAIL_PASSWORD:-my_default_password}
sentry:
  dsn: ${VAR_NOT_SET}
```

Then load your config with *Piny*:

```python
from piny import YamlLoader

config = YamlLoader(path="config.yaml").load()
print(config)
# {'db': {'login': 'user', 'password': 'my_db_password'},
# 'mail': {'login': 'user', 'password': 'my_default_password'},
# 'sentry': {'dsn': None}}
```

CHAPTER

TWO

CLI UTILITY

Piny's also got a command line tool working both with files and standard input and output:

```
$ export PASSWORD=mySecretPassword
$ echo "db: \${PASSWORD}" | piny
db: mySecretPassword
```

CHAPTER

THREE

RATIONALE

Piny allows you to maintain healthy security/convenience balance when it comes to application's configuration. Piny combines readability and versioning you get when using config files, and security that environment variables provide.

Read more about this approach in the blog post.

3.1 Installation

3.1.1 pip

Just use:

```
pip install -U piny
```

Piny supports a few third-party validation libraries (see *Validators*). You may install *Piny* with one of them as en extra requirement:

```
pip install -U 'piny[pydantic]'
```

The full list of extra validation libraries is the following:

- `marshmallow`
- `pydantic`
- `trafaret`

3.1.2 GitHub

You can also clone *Piny* from GitHub and install it using `make install` (see *Contributing to Piny*):

```
git clone https://github.com/pilosus/piny
cd piny
make install
```

3.2 Usage

Piny loads your YAML configuration file. It optionally validates data loaded from config file. *Piny* main logic is in a loader class. You can pass arguments in the loader class to change the way YAML file is parsed and validated.

3.2.1 Loaders

`YamlLoader` loader class is dedicated for use in Python applications. Based on PyYAML, it parses YAML files, (arguably) the most beautiful file format for configuration files!

Basic loader usage is the following.

1. Set your environment variables
2. Mark up your YAML configuration file with these env names:

```
db:
  login: user
  password: ${DB_PASSWORD}
mail:
  login: user
  password: ${MAIL_PASSWORD:-my_default_password}
sentry:
  dsn: ${VAR_NOT_SET}
```

3. In your app load config with *Piny*:

```
from piny import YamlLoader

config = YamlLoader(path="config.yaml").load()
print(config)
# {'db': {'login': 'user', 'password': 'my_db_password'},
# 'mail': {'login': 'user', 'password': 'my_default_password'},
# 'sentry': {'dsn': None}}
```

`YamlStreamLoader` class primary use is Piny CLI tool (see *Command line utility*). But it also can be used interchargably with `YamlLoader` whenever IO streams are used instead of file paths.

class piny.loaders.**YamlLoader**(*path: str, *, matcher: ~typing.Type[~piny.matchers.Matcher] = <class 'piny.matchers.MatcherWithDefaults'>, validator: ~typing.Type[~piny.validators.Validator] | None = None, schema: ~typing.Any = None, **schema_params*)

Bases: `object`

YAML configuration file loader

load(***params*) → Any

Return Python object loaded (optionally validated) from the YAML-file

Parameters

params – named arguments used as optional loading params in validation

class piny.loaders.**YamlStreamLoader**(*stream: str | ~typing.IO[str], *, matcher: ~typing.Type[~piny.matchers.Matcher] = <class 'piny.matchers.MatcherWithDefaults'>, validator: ~typing.Type[~piny.validators.Validator] | None = None, schema: ~typing.Any = None, **schema_params*)

Bases: *YamlLoader*

YAML configuration loader for IO streams, e.g. file objects or stdin

load(***params*) → Any

Return Python object loaded (optionally validated) from the YAML-file

Parameters

params – named arguments used as optional loading params in validation

3.2.2 Matchers

In the *Loaders* section we used Bash-style environment variables with defaults. You may want to discourage such envs in your project. This is where *matchers* come in handy. They apply a regular expression when parsing your YAML file that matches environment variables we want to interpolate.

By default `MatcherWithDefaults` is used. `StrictMatcher` is another matcher class used for plain vanilla envs with no default values support.

Both strict and default matchers return `None` value if environment variable matched is not set in the system.

Basic usage example is the following:

```
from piny import YamlLoader, StrictMatcher

config = YamlLoader(path="config.yaml", matcher=StrictMatcher).load()
```

class piny.matchers.**Matcher**(*stream*)

Bases: `SafeLoader`

Base class for matchers

Use this class only to derive new child classes

static constructor(*loader*, *node*)

matcher: Pattern[str] = re.compile('')

class piny.matchers.**MatcherWithDefaults**(*stream*)

Bases: *Matcher*

Expand an environment variable with its value

Forms supported: ${VAR}, ${VAR:-default} If value is not set and no default value given return None.

static constructor(*loader*, *node*)

matcher: Pattern[str] = re.compile('\\$\\{([a-zA-Z_$0-9]+)(:-.*)?\\}')

class piny.matchers.**StrictMatcher**(*stream*)

Bases: *Matcher*

Expand an environment variable of form ${VAR} with its value

If value is not set return None.

static constructor(*loader*, *node*)

matcher: Pattern[str] = re.compile('\\$\\{([^}^{^:]+)\\}')

3.2.3 Validators

Piny supports *optional* data validation using third-party libraries: Marshmallow, Pydantic, Trafaret.

In order to use data validation pass `validator` and `schema` arguments in the *Loaders* class. You may also initialize loader class with optional named arguments that will be passed to the validator's schema. Additional loading arguments may be passed in `load` method invocation.

class `piny.validators.`**`MarshmallowValidator`**(*schema: Any*, ***params*)

Bases: *`Validator`*

Validator class for Marshmallow library

`load`(*data: Dict[str, Any] | List[Any]*, ***params*)

Load data, return validated data or raise en error

class `piny.validators.`**`PydanticV2Validator`**(*schema: Any*, ***params*)

Bases: *`Validator`*

Validator class for Pydantic Version 2

`load`(*data: Dict[str, Any] | List[Any]*, ***params*)

Load data, return validated data or raise en error

class `piny.validators.`**`PydanticValidator`**(*schema: Any*, ***params*)

Bases: *`Validator`*

Validator class for Pydantic Version 1

`load`(*data: Dict[str, Any] | List[Any]*, ***params*)

Load data, return validated data or raise en error

class `piny.validators.`**`TrafaretValidator`**(*schema: Any*, ***params*)

Bases: *`Validator`*

Validator class for Trafaret library

`load`(*data: Dict[str, Any] | List[Any]*, ***params*)

Load data, return validated data or raise en error

class `piny.validators.`**`Validator`**(*schema: Any*, ***params*)

Bases: `ABC`

Abstract base class for optional validator classes

Use only to derive new child classes, implement all abstract methods

abstract `load`(*data: Dict[str, Any] | List[Any]*, ***params*)

Load data, return validated data or raise en error

Marshmallow validation example

```
import marshmallow as ma
from piny import MarshmallowValidator, StrictMatcher, YamlLoader


class DBSchema(ma.Schema):
    login = ma.fields.String(required=True)
```

(continues on next page)

(continued from previous page)

```
    password = ma.fields.String()


class ConfigSchema(ma.Schema):
    db = ma.fields.Nested(DBSchema)


config = YamlLoader(
    path="database.yaml",
    matcher=StrictMatcher,
    validator=MarshmallowValidator,
    schema=ConfigSchema,
).load(many=False)
```

Pydantic validation example

```
from pydantic import BaseModel
from piny import PydanticV2Validator, StrictMatcher, YamlLoader

# Watch out!
# Pydantic V2 deprecated some model's methods:
# https://docs.pydantic.dev/2.0/migration/
#
# For Pydantic v2 use `PydanticV2Validator`
# For Pydantic v1 use `PydanticValidator`

class DBModel(BaseModel):
    login: str
    password: str


class ConfigModel(BaseModel):
    db: DBModel


config = YamlLoader(
    path="database.yaml",
    matcher=StrictMatcher,
    validator=PydanticV2Validator,
    schema=ConfigModel,
).load()
```

Trafaret validation example

```
import trafaret
from piny import TrafaretValidator, StrictMatcher, YamlLoader


DBSchema = trafaret.Dict(login=trafaret.String, password=trafaret.String)
ConfigSchema = trafaret.Dict(db=DBSchema)

config = YamlLoader(
    path="database.yaml",
    matcher=StrictMatcher,
    validator=TrafaretValidator,
    schema=ConfigSchema,
).load()
```

3.2.4 Exceptions

`LoadingError` is thrown when something goes wrong with reading or parsing a YAML file. `ValidationError` is a wrapper for exceptions raised by the libraries for optional data validation. Original exception can be accessed by `origin` attribute. It comes in handy when you need more than just an original exception message (e.g. a dictionary of validation errors).

Both exceptions inherit from the `ConfigError`.

exception `piny.errors.`**`ConfigError`**(*origin: Exception | None = None*, ***context: Any*)

Bases: *`PinyErrorMixin`*, `Exception`

Base class for Piny exceptions

exception `piny.errors.`**`LoadingError`**(*origin: Exception | None = None*, ***context: Any*)

Bases: *`ConfigError`*

Exception for reading or parsing configuration file errors

`msg_template: str = 'Loading YAML file failed: {reason}'`

class `piny.errors.`**`PinyErrorMixin`**(*origin: Exception | None = None*, ***context: Any*)

Bases: `object`

Mixin class to wrap and format original exception

`msg_template: str`

exception `piny.errors.`**`ValidationError`**(*origin: Exception | None = None*, ***context: Any*)

Bases: *`ConfigError`*

Exception for data validation errors

`msg_template: str = 'Validation failed: {reason}'`

3.2.5 Command line utility

Piny comes with CLI tool that substitutes the values of environment variables in input file or `stdin` and write result to an output file or `stdout`. Piny CLI utility is somewhat similar to `GNU/gettext` envsubst but works with files too.

piny

Substitute environment variables with their values.

Read INPUT, find environment variables in it, substitute them with their values and write to OUTPUT.

INPUT and OUTPUT can be files or standard input and output respectively. With no INPUT, or when INPUT is -, read standard input. With no OUTPUT, or when OUTPUT is -, write to standard output.

Examples:

piny input.yaml output.yaml
piny - output.yaml
piny input.yaml -
tail -n 12 input.yaml | piny > output.yaml

```
piny [OPTIONS] [INPUT] [OUTPUT]
```

Options

`--strict, --no-strict`
Enable or disable strict matcher

Arguments

`INPUT`
Optional argument

`OUTPUT`
Optional argument

3.3 Integration Examples

3.3.1 Flask

Flask is a microframework for Python web applications. It's flexible and extensible. Although there are best practices and traditions, Flask doesn't really enforce the only one way to do it.

If you are working on a small project the chances are that you are using some Flask extensions like Flask-Mail or Flask-WTF. The extensions of the past are often got configured through environment variables only. It makes the use of *Piny* cumbersome. In mid-sized and large Flask projects though, you usually avoid using extra dependencies whenever possible. In such a case you can fit your code to use *Piny* pretty easy.

Here is an example of a simple Flask application. Configuration file is loaded with *Piny* and validated with *Pydantic*.

```python
from flask import Flask
from flask.logging import default_handler
from piny import YamlLoader, StrictMatcher, PydanticV2Validator
from pydantic import BaseModel, validator
from typing import Any, Dict, Optional
from werkzeug.serving import run_simple
import logging
import sys

# Watch out!
# Pydantic V2 deprecated some model's methods:
# https://docs.pydantic.dev/2.0/migration/
#
# For Pydantic v2 use `PydanticV2Validator`
# For Pydantic v1 use `PydanticValidator`


#
# Validation
#


class AppSettings(BaseModel):
    company: str
    secret: str
    max_content_len: Optional[int] = None
    debug: bool = False
    testing: bool = False


class LoggingSettings(BaseModel):
    fmt: str
    date_fmt: str
    level: str

    @validator("level")
    def validate_name(cls, value):
        upper = value.upper()
        if upper not in logging._nameToLevel:
            raise ValueError("Invalid logging level")
        return upper


class Configuration(BaseModel):
    app: AppSettings
    logging: LoggingSettings


#
# Helpers
#
```

(continues on next page)

(continued from previous page)

```
def configure_app(app: Flask, configuration: Dict[str, Any]) -> None:
    """
    Apply configs to application
    """
    app.settings = configuration
    app.secret_key = app.settings["app"]["secret"].encode("utf-8")


def configure_logging(app: Flask) -> None:
    """
    Configure app's logging
    """
    app.logger.removeHandler(default_handler)
    log_formatter = logging.Formatter(
        fmt=app.settings["logging"]["fmt"], datefmt=app.settings["logging"]["date_fmt"]
    )
    log_handler = logging.StreamHandler()
    log_handler.setFormatter(log_formatter)
    log_handler.setLevel(app.settings["logging"]["level"])
    app.logger.addHandler(log_handler)


#
# Factory
#


def create_app(path: str) -> Flask:
    """
    Application factory
    """
    # Get and validate config
    config = YamlLoader(
        path=path,
        matcher=StrictMatcher,
        validator=PydanticV2Validator,
        schema=Configuration,
    ).load()

    # Initialize app
    app = Flask(__name__)

    # Configure app
    configure_app(app, config)
    configure_logging(app)

    return app


if __name__ == "__main__":
    app = create_app(sys.argv[1])
```

(continues on next page)

(continued from previous page)

```
    @app.route("/")
    def hello():
        return "Hello World!"

    # Run application:
    # $ python flask_integration.py your-config.yaml
    run_simple(hostname="localhost", port=5000, application=app)
```

You can use the same pattern with application factory in other frameworks, like aiohttp or sanic.

3.3.2 Command line

There are many possible applications for *Piny* CLI utility. For example, you can use it for Kubernetes deployment automation in your CI/CD pipeline.

Piny command line tool works both with standard input/output and files.

Standard input and output

```
$ export PASSWORD=mySecretPassword
$ echo "db: \${PASSWORD}" | piny
db: mySecretPassword
```

Files

```
$ piny config.template config.yaml
```

Or you can substitute environment variables in place:

```
$ piny production.yaml production.yaml
```

3.4 Best practices

- Maintain a healthy security/convenience balance for your config
- Mark up entity as an environment variable in your YAML if and only if it really is a *secret* (login/passwords, private API keys, crypto keys, certificates, or maybe DB hostname too? You decide)
- When loading config file, validate your data. Piny supports a few popular data validation tools.
- Store your config files in the version control system along with your app's code.
- Environment variables are set by whoever is responsible for the deployment. Modern orchestration systems like Kubernetes make it easier to keep envs secure (see Kubernetes Secrets).

3.5 Contributing to Piny

Piny is a proof-of-concept. It's developed specifically (but not limited to!) for the containerized Python applications deployed with orchestration systems like `docker compose` or `Kubernetes`.

Piny is still in its early stage of development. The API may change, backward compatibility between minor versions is not guaranteed until version 1.0.0 is reached.

Piny sticks to the Unix-way's rule *Do One Thing and Do It Well*. Piny is all about interpolating environment variables in configuration files. Other features like YAML-parsing or data validation are implemented using third-party libraries whenever possible.

You are welcome to contribute to *Piny* as long as you follow the rules.

3.5.1 General rules

1. Before writing any *code* take a look at the existing open issues. If none of them is about the changes you want to contribute, open up a new issue. Fixing a typo requires no issue though, just submit a Pull Request.
2. If you're looking for an open issue to fix, check out labels help wanted and good first issue on GitHub.
3. If you plan to work on an issue open not by you, write about your intention in the comments *before* you start working.
4. Follow an Issue/Pull Request template.

3.5.2 Development rules

1. Fork Piny on GitHub.
2. Clone your fork with `git clone`.
3. Use `Python 3.6+`, `git`, `make` and `virtualenv`.
4. Create and activate `virtualenv`.
5. Install *Piny* and its dependencies with `make install`.
6. Follow GitHub Flow: create a new branch from `master` with `git checkout -b <your-feature-branch>`. Make your changes.
7. Fix your code's formatting and imports with `make format`.
8. Run unit-tests and linters with `make check`.
9. Build documentation with `make docs`.
10. Commit, push, open new Pull Request.
11. Make sure Travis CI/CD pipeline succeeds.

3.6 Changelog

3.6.1 v1.1.0 (2023-09-22)

- Added: new validator *PydanticV2Validator* to support Pydantic v2

3.6.2 v1.0.2 (2023-02-03)

- Update GitHub workflow for CI: run tests & license checks for PRs, pushes to master and tags (#202) by @pilosus
- Make dependabot update GitHub Actions (#202) by @pilosus

3.6.3 v1.0.1 (2023-02-03)

- Run tests against locally installed package instead of using ugly imports (#200) by @pilosus

3.6.4 v1.0.0 (2023-01-02)

See release notes to *v1.0.0rc1*

3.6.5 v1.0.0rc1 (2023-01-01)

Release breaks backward compatibility!

- Bump major dependencies: *PyYAML>=6,<7 Click>=8,<9* (#192) by @pilosus
- *Marshmallow* integration supports only v3.0.0 and later (#192) by @pilosus
- Move to *pyproject.toml* for packaging (#193) by @pilosus
- Raise Python requirement to *>=3.7* (#193) by @pilosus

3.6.6 v0.6.0 (2019-06-27)

- Add CLI utility (#35) by @pilosus
- Update documentation, add integration examples (#34) by @pilosus

3.6.7 v0.5.2 (2019-06-17)

- Fix `Help` section in `README.rst` (#31) by @pilosus
- Fix Sphinx release variable (#30) by @pilosus

3.6.8 v0.5.1 (2019-06-17)

- Fix Sphinx config, fix README.rst image markup (#28) by @pilosus

3.6.9 v0.5.0 (2019-06-17)

- Sphinx documentation added (#12) by @pilosus
- Piny artwork added (#6) by Daria Runenkova and @pilosus

3.6.10 v0.4.2 (2019-06-17)

- Rename parent exception `PinyError` to `ConfigError` (#18) by @pilosus
- Add feature request template for GitHub Issues (#20) by @pilosus

3.6.11 v0.4.1 (2019-06-17)

- Issue and PR templates added, minor docs fixes (#16) by @pilosus

3.6.12 v0.4.0 (2019-06-16)

- Data validators support added for `Pydantic`, `Marshmallow` (#2) by @pilosus
- `CONTRIBUTING.rst` added (#4) by @pilosus

3.6.13 v0.3.1 (2019-06-09)

- Minor RST syntax fix in README.rst (#9) by @pilosus

3.6.14 v0.3.0 (2019-06-09)

- README.rst extended with `Rationale` and `Best practices` sections (#5) by @pilosus

3.6.15 v0.2.0 (2019-06-09)

- StrictMatcher added (#3) by @pilosus

3.6.16 v0.1.1 (2019-06-07)

- CI/CD config minor tweaks
- README updated

3.6.17 v0.1.0 (2019-06-07)

- YamlLoader added
- Makefile added
- CI/CD minimal pipeline added

3.6.18 v0.0.1 (2019-06-07)

- Start the project

3.7 Fun facts

Piny is a recursive acronym for *Piny Is Not YAML*. Not only it's a library name, but also a name for YAML marked up with environment variables.

PYTHON MODULE INDEX

p

INDEX

Symbols

C

I

L

M

O

P

S

T

V

Y