Yapconf Documentation

Release 0.2.3

Logan Asher Jones

Apr 16, 2018

Contents

Contents:

CHAPTER 1

Yapconf

Yet Another Python Configuration. A simple way to manage configurations for python applications.

Yapconf allows you to easily manage your python application's configuration. It handles everything involving your application's configuration. Often times exposing your configuration in sensible ways can be difficult. You have to consider loading order, and lots of boilerplate code to update your configuration correctly. Now what about CLI support? Migrating old configs to the new config? Yapconf can help you.

1.1 Features

Yapconf helps manage your python application's configuration

- JSON/YAML config file support
- Argparse integration
- Environment Loading
- Bootstrapping
- Migrate old configurations to new configurations

1.2 Quick Start

To install Yapconf, run this command in your terminal:

```
$ pip install yapconf
```

Then you can use Yapconf yourself!

```
from yapconf import YapconfSpec

# First define a specification
```

(continues on next page)

(continued from previous page)

```
my_spec = YapconfSpec({"foo": {"type": "str", "default": "bar"}}, env_prefix='MY_APP_
↪')

# Then load the configuration in whatever order you want!
# load_config will automatically look for the 'foo' value in
# '/path/to/config.yml', then the environment, finally
# falling back to the default if it was not found elsewhere
config = my_spec.load_config('/path/to/config.yml', 'ENVIRONMENT')

print(config.foo)
print(config['foo'])
```

You can also add these arguments to the command line very easily

```
import argparse

parser = argparse.ArgumentParser()

# This will add --foo as an argument to your python program
my_spec.add_arguments(parser)

cli_args = vars(parser.parse_args(sys.argv[1:]))

# Now you can load these via load_config:
config = my_spec.load_config(cli_args, '/path/to/config.yml', 'ENVIRONMENT')
```

For more detailed information and better walkthroughs, checkout the documentation!

1.3 Documentation

Documentation is available at https://yapconf.readthedocs.io

1.4 Credits

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

CHAPTER 2

Installation

2.1 Stable release

To install Yapconf, run this command in your terminal:

```
$ pip install yapconf
```

This is the preferred method to install Yapconf, 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 Yapconf can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/loganasherjones/yapconf
```

Or download the tarball:

```
$ curl  -OL https://github.com/loganasherjones/yapconf/tarball/master
```

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

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

CHAPTER 3

Usage

In order to use Yapconf in a project, you will first need to create your specification object. There are lots of options for this object, so we'll just start with the basics. Check out the *Item Arguments* section for all the options available to you. For now, let's just assume we have the following specification defined

```
from yapconf import YapconfSpec

my_spec = YapconfSpec({
    'db_name': {'type': 'str'},
    'db_port': {'type': 'int'},
    'db_host': {'type': 'str'},
    'verbose': {'type': 'bool', 'default': True},
    'filename': {'type': 'str'},
})
```

Now that you have a specification for your configuration, you can load your config from lots of different places using `load_config`. When using this method, it is significant the order in which you pass your arguments as it sets the precedence for load order. Let's see this in practice:

```
# Let's say you loaded this dict from the command-line (more on that later)
cli_args = {'filename': '/path/to/config', 'db_name': 'db_from_cli'}

# Also assume you have /some/config.yml that has the following:
#   db_name: db_from_config_file
#   db_port: 1234
config_file = '/some/config.yml' # JSON is also supported!

# Finally, let's assume you have the following set in your environment
# DB_NAME="db_from_environment"
# FILENAME="/some/default/config.yml"
# DB_HOST="localhost"

# You can load your config:
config = my_spec.load_config(cli_args, config_file, 'ENVIRONMENT')
```

(continues on next page)

(continued from previous page)

```
# You now have a config object which can be accessed via attributes or keys:
config.db_name # > db_from_cli
config['db_port'] # > 1234
config.db_host # > localhost
config['verbose'] # > True
config.filename # > /path/to/config

# If you loaded in a different order, you'll get a different result
config = my_spec.load_config('ENVIRONMENT', config_file, cli_args)
config.db_name # > db_from_environment
```

This config object is powered by python-box which is a handy utility for handling your config object. It behaves just like a dictionary and you can treat it as such!

3.1 Nested Items

In a lot of cases, it makes sense to nest your configuration, for example, if we wanted to take all of our database configuration and put it into a single dictionary, that would make a lot of sense. You would specify this to yapconf as follows:

```
nested_spec = YapconfSpec({
    'db': {
        'type': 'dict',
        'items': {
            'name': { 'type': 'str' },
            'port': { 'type': 'int' }
        }
    }
})

config = nested_spec.load_config({'db': {'name': 'db_name', 'port': 1234}})

config.db.name # returns 'name'
config.db.port # returns 1234
config.db # returns the db dictionary
```

3.2 List Items

List items are a special class of nested items which is only allowed to have a single item listed. It can be specified as follows:

```
list_spec = YapconfSpec({
    'names': {
        'type': 'list',
        'items': {
            'name': {'type': 'str'}
        }
    }
})
```

(continues on next page)

(continued from previous page)

```
config = list_spec.load_config({'names': ['a', 'b', 'c']})

config.names # returns ['a', 'b', 'c']
```

3.3 Environment Loading

If no `env_name` is specified for each item, then by default, Yapconf will automatically format the item's name to be all upper-case and snake case. So the name `foo_bar` will become `FOO_BAR` and `fooBar` will become `FOO_BAR`. If you do not want to apply this formatting, set `format_env` to `False`. Loading `list` items and `dict` items from the environment is not supported and as such `env_name` s that are set for these items will be ignored.

Often times, you will want to prefix environment variables with your application name or something else. You can set an environment prefix on the `YapconfSpec` item via the `env_prefix`:

```
import os

env_spec = Specification({'foo': {'type': 'str'}}, 'MY_APP_')

os.environ['FOO'] = 'not_namespaced'
os.environ['MY_APP_FOO'] = 'namespaced_value'

config = env_spec.load_config('ENVIRONMENT')

config.foo # returns 'namespaced_value'
```

Note: When using an `env_name` with `env_prefix` the `env_prefix` will still be applied to the name you provided. If you want to avoid this behavior, set the `apply_env_prefix` to `False`.

As of version 0.1.2, you can specify additional environment names via: `alt_env_names`. The `apply_env_prefix` flag will also apply to each of these. If your environment names collide with other names, then an error will get raised when the specification is created.

3.4 CLI Support

Yapconf has some great support for adding your configuration items as command-line arguments by utilizing argparse. Let's assume the `my_spec` object from the original example

```
import argparse

my_spec = YapconfSpec({
    'db_name': {'type': 'str'},
    'db_port': {'type': 'int'},
    'db_host': {'type': 'str'},
    'verbose': {'type': 'bool', 'default': True},
    'filename': {'type': 'str'},
})

parser = argparser.ArgumentParser()
my_spec.add_arguments(parser)
```

(continues on next page)

(continued from previous page)

```
args = [
    '--db-name', 'db_name',
    '--db-port', '1234',
    '--db-host', 'localhost',
    '--no-verbose',
    '--filename', '/path/to/file'
]

cli_values = vars(parser.parse_args(args))

config = my_spec.load_config(cli_values)

config.db_name # 'db_name'
config.db_port # 1234
config.db_host # 'localhost'
config.verbose # False
config.filename # '/path/to/file'
```

Yapconf makes adding CLI arguments very easy! If you don't want to expose something over the command line you can set the `cli_expose` flag to `False`.

3.4.1 Boolean Items and the CLI

Boolean items will add special flags to the command-line based on their defaults. If you have a default set to `True` then a `--no-{item_name}` flag will get added. If the default is `False` then a `--{{item_name}}` will get added as an argument. If no default is specified, then both will be added as mutually exclusive arguments.

3.4.2 Nested Items and the CLI

Yapconf even supports `list` and `dict` type items from the command-line:

```
import argparse

spec = YapconfSpec({
    'names': {
        'type': 'list',
        'items': {
            'name': {'type': 'str'}
        }
    },
    'db': {
        'type': 'dict',
        'items': {
            'host': {'type': 'str'},
            'port': {'type': 'int'}
        },
    }
})

parser = argparse.ArgumentParser()

cli_args = [
    '--name', 'foo',
    '--name', 'bar',
```

(continues on next page)

(continued from previous page)

```
    '--db-host', 'localhost',
    '--db-port', '1234',
    '--name', 'baz'
]

cli_values = vars(parser.parse_args(args))

config = my_spec.load_config(cli_values)

config.names # ['foo', 'bar', 'baz']
config.db.host # 'localhost'
config.db.port # 1234
```

3.4.3 Limitations

There are a few limitations to how far down the rabbit-hole Yapconf is willing to go. Yapconf does not support `list` type items with either `dict` or `list` children. The reason is that it would be very cumbersome to start specifying which items belong to which dictionaries and in which index in the list.

3.4.4 CLI/Environment Name Formatting

A quick note on formatting and `yapconf`. Yapconf tries to create sensible ways to convert your config items into "normal" environment variables and command-line arguments. In order to do this, we have to make some assumptions about what "normal" environment variables and command-line arguments are.

By default, environment variables are assumed to be all upper-case, snake-case names. The item name `foO_BaR` would become `FOO_BAR` in the environment.

By default, command-line argument are assumed to be kebab-case. The item name `foo_bar` would become `--foo-bar`

If you do not like this formatting, then you can turn it off by setting the `format_env` and `format_cli` flags.

3.5 Config Migration

Throughout the lifetime of an application it is common to want to move configuration around, changing both the names of configuration items and the default values for each. Yapconf also makes this migration a breeze! Each item has a `previous_defaults` and `previous_names` values that can be specified. These values help you migrate previous versions of config files to newer versions. Let's see a basic example where we might want to update a config file with a new default:

```
# Assume we have a JSON config file ('/path/to/config.json') like the following:
# {"db_name": "test_db_name", "db_host": "1.2.3.4"}

spec = YapconfSpec({
    'db_name': {'type': 'str', 'default': 'new_default', 'previous_defaults': ['test_
↪db_name']},
    'db_host': {'type': 'str', 'previous_defaults': ['localhost']}
})

# We can migrate that file quite easily with the spec object:
spec.migrate_config_file('/path/to/config.json')
```

(continues on next page)

(continued from previous page)

```
# Will result in /path/to/config.json being overwritten:
# {"db_name": "new_default", "db_host": "1.2.3.4"}
```

You can specify different output config files also:

```
spec.migrate_config_file('/path/to/config.json',
                         output_file_name='/new/path/to/config.json')
```

There are many values you can pass to `migrate_config_file`, by default it looks like this:

```
spec.migrate_config_file('/path/to/config',
                         always_update=False,     # Always update values (even if you␣
↪set them to None)
                         current_file_type=None, # Used for transitioning between␣
↪json and yaml config files
                         output_file_name=None,   # Will default to current file name
                         output_file_type=None,   # Used for transitioning between␣
↪json and yaml config files
                         create=True,             # Create the file if it doesn't exist
                         update_defaults=True     # Update the defaults
                         )
```

3.6 YAML Support

Yapconf knows how to output and read both `json` and `yaml` files. However, to keep the dependencies to a minimum it does not come with `yaml`. You will have to manually install either `pyyaml` or `ruamel.yaml` if you want to use `yaml`.

3.7 Item Arguments

For each item in a specification, you can set any of these keys:

Name	Default	Description
name	N/A	The name of the config item
item_type	`'str'`	The python type of the item (`'str'`, `'int'`, `'long'`, `'float'`, `'bool'`, `'complex'`, `'dict'`, `'list'`)
default	`None`	The default value for this item
env_name	`name.upper()`	The name to search in the environment
description	`None`	Description of the item
required	`True`	Specifies if the item is required to exist
cli_short_name	`None`	One-character command-line shortcut
cli_choices	`None`	List of possible values for the item from the command-line
previous_names	`None`	List of previous names an item had
previous_defaults	`None`	List of previous defaults an item had
items	`None`	Nested item definition for use by `list` or `dict` type items
cli_expose	`True`	Specifies if this item should be added to arguments on the command-line (nested `list` are always `False`)
separator	`.`	The separator to use for `dict` type items (useful for `previous_names`)
bootstrap	`False`	A flag that indicates this item needs to be loaded before others can be loaded
format_env	`True`	A flag to determine if environment variables will be all upper-case SNAKE_CASE.
format_cli	`True`	A flag to determine if we should format the command-line arguments to be kebab-case.
apply_env_prefix	`True`	Apply the env_prefix even if the environment name was set manually. Ignored if `format_env` is `False`
choices	`None`	A list of valid choices for the item. Cannot be set for `dict` items.
alt_env_names	`[]`	A list of alternate environment names.

CHAPTER 4

yapconf

4.1 yapconf package

4.1.1 Submodules

4.1.2 yapconf.actions module

class yapconf.actions.**AppendBoolean**(*option_strings*, *dest*, *const*, *default=None*, *required=False*, *help=None*, *metavar=None*)

Bases: argparse.Action

Action used for appending boolean values on the command-line

class yapconf.actions.**AppendReplace**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: argparse.Action

argparse.Action used for appending values on the command-line

class yapconf.actions.**MergeAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*, *child_action=None*, *separator='.'*, *child_const=None*)

Bases: argparse.Action

Merges command-line values into a single dictionary based on separator.

Each MergeAction has a child_action that indicates what should happen for each value. It uses the separator to determine the eventual location for each of its values.

The dest is split up by separator and each string is in turn used to determine the key that should be used to store this value in the dictionary that will get created.

child_action
The action that determines which value is stored

child_const
For booleans, this is the value used

separator
A separator to split up keys in the dictionary

4.1.3 yapconf.exceptions module

yapconf.exceptions

This module contains the set of Yapconf's exceptions.

exception yapconf.exceptions.**YapconfDictItemError**
Bases: *yapconf.exceptions.YapconfItemError*

There was an error creating a YapconfDictItem from the specification

exception yapconf.exceptions.**YapconfError**
Bases: Exception

There was an error while handling your config

exception yapconf.exceptions.**YapconfItemError**
Bases: *yapconf.exceptions.YapconfError*

There was an error creating a YapconfItem from the specification

exception yapconf.exceptions.**YapconfItemNotFound**(*message*, *item*)
Bases: *yapconf.exceptions.YapconfItemError*

We searched through all the overrides and could not find the item

exception yapconf.exceptions.**YapconfListItemError**
Bases: *yapconf.exceptions.YapconfItemError*

There was an error creating a YapconfListItem from the specification

exception yapconf.exceptions.**YapconfLoadError**
Bases: *yapconf.exceptions.YapconfError*

There was an error while trying to load the overrides provided

exception yapconf.exceptions.**YapconfSpecError**
Bases: *yapconf.exceptions.YapconfError*

There was an error detected in the specification provided

exception yapconf.exceptions.**YapconfValueError**
Bases: *yapconf.exceptions.YapconfItemError*

We found an item in the overrides but it wasn't what we expected

4.1.4 yapconf.items module

class yapconf.items.**YapconfBoolItem**(*name*, *item_type='bool'*, *default=None*, *env_name=None*, *description=None*, *required=True*, *cli_short_name=None*, *cli_choices=None*, *previous_names=None*, *previous_defaults=None*, *children=None*, *cli_expose=True*, *separator='.'*, *prefix=None*, *bootstrap=False*, *format_cli=True*, *format_env=True*, *env_prefix=None*, *apply_env_prefix=True*, *choices=None*, *alt_env_names=None*)

Bases: *yapconf.items.YapconfItem*

A YapconfItem specifically for Boolean behavior

FALSY_VALUES = ('n', 'no', 'f', 'false', '0', 0, False)

TRUTHY_VALUES = ('y', 'yes', 't', 'true', '1', 1, True)

add_argument(*parser*, *bootstrap=False*)

Add boolean item as an argument to the given parser.

An exclusive group is created on the parser, which will add a boolean-style command line argument to the parser.

Examples

A non-nested boolean value with the name 'debug' will result in a command-line argument like the following:

'–debug/–no-debug'

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to add this item to.
- **bootstrap** (*bool*) – Flag to indicate whether you only want to mark this item as required or not.

convert_config_value(*value*, *label*)

Converts all 'Truthy' values to True and 'Falsy' values to False.

Parameters

- **value** – Value to convert
- **label** – Label of the config which this item was found.

Returns:

class yapconf.items.**YapconfDictItem**(*name*, *item_type='dict'*, *default=None*, *env_name=None*, *description=None*, *required=True*, *cli_short_name=None*, *cli_choices=None*, *previous_names=None*, *previous_defaults=None*, *children=None*, *cli_expose=True*, *separator='.'*, *prefix=None*, *bootstrap=False*, *format_cli=True*, *format_env=True*, *env_prefix=None*, *apply_env_prefix=True*, *choices=None*, *alt_env_names=None*)

Bases: *yapconf.items.YapconfItem*

A YapconfItem for capture dict-specific behavior

add_argument(*parser*, *bootstrap=False*)
Add dict-style item as an argument to the given parser.

The dict item will take all the nested items in the dictionary and namespace them with the dict name, adding each child item as their own CLI argument.

Examples

A non-nested dict item with the name 'db' and children named 'port' and 'host' will result in the following being valid CLI args:

['–db-host', 'localhost', '–db-port', '1234']

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to add this item to.
- **bootstrap** (*bool*) – Flag to indicate whether you only want to mark this item as required or not.

convert_config_value(*value*, *label*)

get_config_value(*overrides*)
Get the configuration value from all overrides.

Iterates over all overrides given to see if a value can be pulled out from them. It will convert each of these values to ensure they are the correct type.

Parameters **overrides** – A list of tuples where each tuple is a label and a dictionary representing a configuration.

Returns The converted configuration value.

Raises

- YapconfItemNotFound – If an item is required but could not be found in the configuration.
- YapconfItemError – If a possible value was found but the type cannot be determined.
- YapconfValueError – If a possible value is found but during conversion, an exception was raised.

migrate_config(*current_config*, *config_to_migrate*, *always_update*, *update_defaults*)
Migrate config value in current_config, updating config_to_migrate.

Given the current_config object, it will attempt to find a value based on all the names given. If no name could be found, then it will simply set the value to the default.

If a value is found and is in the list of previous_defaults, it will either update or keep the old value based on if update_defaults is set.

If a non-default value is set it will either keep this value or update it based on if always_update is true.

Parameters

- **current_config** (*dict*) – Current configuration.
- **config_to_migrate** (*dict*) – Config to update.
- **always_update** (*bool*) – Always update value.
- **update_defaults** (*bool*) – Update values found in previous_defaults

class yapconf.items.**YapconfItem**(*name*, *item_type='str'*, *default=None*, *env_name=None*, *description=None*, *required=True*, *cli_short_name=None*, *cli_choices=None*, *previous_names=None*, *previous_defaults=None*, *children=None*, *cli_expose=True*, *separator='.'*, *prefix=None*, *bootstrap=False*, *format_cli=True*, *format_env=True*, *env_prefix=None*, *apply_env_prefix=True*, *choices=None*, *alt_env_names=None*)

Bases: `object`

A simple configuration item for interacting with configurations.

A `YapconfItem` represent the following types: (`str`, `int`, `long`, `float`, `complex`). It also acts as the base class for the other `YapconfItem` types. It provides several basic functions. It helps create CLI arguments to be used by `argparse.ArgumentParser`. It also makes getting a particular configuration value simple.

In general this class is expected to be used by the `YapconfSpec` class to help manage your configuration.

name
str – The name of the config value.

item_type
str – The type of config value you are expecting.

default
The default value if no configuration value can be found.

env_name
The name to search in the environment.

description
The description of your configuration item.

required
Whether or not the item is required to be present.

cli_short_name
A short name (1-character) to identify your item on the command-line.

cli_choices
A list of possible choices on the command-line.

previous_names
A list of names that used to identify this item. This is useful for config migrations.

previous_defaults
A list of previous default values given to this item. Again, useful for config migrations.

children
Any children of this item. Not used by this base class.

cli_expose
A flag to indicate if the item should be exposed from the command-line. It is possible for this value to be overwritten based on whether or not this item is part of a nested list.

separator
A separator used to split apart parent names in the prefix.

prefix
A delimited list of parent names

bootstrap
A flag to determine if this item is required for bootstrapping the rest of your configuration.

format_cli
A flag to determine if we should format the command-line arguments to be kebab-case.

format_env
A flag to determine if environment variables will be all upper-case SNAKE_CASE.

env_prefix
The env_prefix to apply to the environment name.

apply_env_prefix
Apply the env_prefix even if the environment name was set manually. Setting format_env to false will override this behavior.

choices
A list of valid choices for the item.

alt_env_names
A list of alternate environment names.

Raises `YapconfItemError` – If any of the information given during initialization results in an invalid item.

add_argument(*parser*, *bootstrap=False*)
Add this item as an argument to the given parser.

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to add this item to.
- **bootstrap** – Flag to indicate whether you only want to mark this item as required or not

all_env_names

convert_config_value(*value*, *label*)

get_config_value(*overrides*)
Get the configuration value from all overrides.

Iterates over all overrides given to see if a value can be pulled out from them. It will convert each of these values to ensure they are the correct type.

Parameters **overrides** – A list of tuples where each tuple is a label and a dictionary representing a configuration.

Returns The converted configuration value.

Raises

- `YapconfItemNotFound` – If an item is required but could not be found in the configuration.
- `YapconfItemError` – If a possible value was found but the type cannot be determined.
- `YapconfValueError` – If a possible value is found but during conversion, an exception was raised.

migrate_config(*current_config*, *config_to_migrate*, *always_update*, *update_defaults*)
Migrate config value in current_config, updating config_to_migrate.

Given the current_config object, it will attempt to find a value based on all the names given. If no name could be found, then it will simply set the value to the default.

If a value is found and is in the list of previous_defaults, it will either update or keep the old value based on if update_defaults is set.

If a non-default value is set it will either keep this value or update it based on if `always_update` is true.

Parameters

- **current_config** (*dict*) – Current configuration.
- **config_to_migrate** (*dict*) – Config to update.
- **always_update** (*bool*) – Always update value.
- **update_defaults** (*bool*) – Update values found in previous_defaults

update_default(*new_default*, *respect_none=False*)

Update our current default with the new_default.

Parameters

- **new_default** – New default to set.
- **respect_none** – Flag to determine if `None` is a valid value.

class yapconf.items.**YapconfListItem**(*name*, *item_type='list'*, *default=None*, *env_name=None*, *description=None*, *required=True*, *cli_short_name=None*, *cli_choices=None*, *previous_names=None*, *previous_defaults=None*, *children=None*, *cli_expose=True*, *separator='.'*, *prefix=None*, *bootstrap=False*, *format_cli=True*, *format_env=True*, *env_prefix=None*, *apply_env_prefix=True*, *choices=None*, *alt_env_names=None*)

Bases: *yapconf.items.YapconfItem*

A YapconfItem for capture list-specific behavior

add_argument(*parser*, *bootstrap=False*)

Add list-style item as an argument to the given parser.

Generally speaking, this works mostly like the normal append action, but there are special rules for boolean cases. See the AppendReplace action for more details.

Examples

A non-nested list value with the name 'values' and a child name of 'value' will result in a command-line argument that will correctly handle arguments like the following:

['–value', 'VALUE1', '–value', 'VALUE2']

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to add this item to.
- **bootstrap** (*bool*) – Flag to indicate whether you only want to mark this item as required or not.

convert_config_value(*value*, *label*)

get_config_value(*overrides*)

Get the configuration value from all overrides.

Iterates over all overrides given to see if a value can be pulled out from them. It will convert each of these values to ensure they are the correct type.

Parameters **overrides** – A list of tuples where each tuple is a label and a dictionary representing a configuration.

Returns The converted configuration value.

Raises

- `YapconfItemNotFound` – If an item is required but could not be found in the configuration.
- `YapconfItemError` – If a possible value was found but the type cannot be determined.
- `YapconfValueError` – If a possible value is found but during conversion, an exception was raised.

`yapconf.items.`**`from_specification`**(*specification*, *env_prefix=None*, *separator='.'*, *parent_names=None*)

Used to create YapconfItems from a specification dictionary.

Parameters

- **`specification`** (*`dict`*) – The specification used to initialize `YapconfSpec`
- **`env_prefix`** (*`str`*) – Prefix to add to environment names
- **`separator`** (*`str`*) – Separator for nested items
- **`parent_names`** (*`list`*) – Parents names of any given item

Returns A dictionary of names to YapconfItems

4.1.5 yapconf.spec module

`class` `yapconf.spec.`**`YapconfSpec`**(*specification*, *file_type='json'*, *env_prefix=None*, *encoding='utf-8'*, *separator='.'*)

Bases: `object`

Object which holds your configuration's specification.

The YapconfSpec item is the main interface into the yapconf package. It will help you load, migrate, update and add arguments for your application.

Examples

```
>>> from yapconf import YapconfSpec
```

First define a specification

```
>>> my_spec = YapconfSpec(
... {"foo": {"type": "str", "default": "bar"}},
... env_prefix='MY_APP_')
```

Then load the configuration in whatever order you want! load_config will automatically look for the 'foo' value in '/path/to/config.yml', then the environment, finally falling back to the default if it was not found elsewhere

```
>>> config = my_spec.load_config('/path/to/config.yml', 'ENVIRONMENT')
>>> print(config.foo)
>>> print(config['foo'])
```

add_arguments(*parser*, *bootstrap=False*)
Adds all items to the parser passed in.

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to add all items to.
- **bootstrap** (*bool*) – Flag to indicate whether you only want to mark bootstrapped items as required on the command-line.

defaults
dict – All defaults for items in the specification.

get_item(*name*, *bootstrap=False*)
Get a particular item in the specification.

Parameters

- **name** (*str*) – The name of the item to retrieve.
- **bootstrap** (*bool*) – Only search bootstrap items

Returns (YapconfItem): A YapconfItem if it is found, None otherwise.

load_config(**args*, ***kwargs*)
Load a config based on the arguments passed in.

The order of arguments passed in as *args is significant. It indicates the order of precedence used to load configuration values. Each argument can be a string, dictionary or a tuple. There is a special case string called 'ENVIRONMENT', otherwise it will attempt to load the filename passed in as a string.

By default, if a string is provided, it will attempt to load the file based on the file_type passed in on initialization. If you want to load a mixture of json and yaml files, you can specify them as the 3rd part of a tuple.

Examples

You can load configurations in any of the following ways:

```
>>> my_spec = YapconfSpec({'foo': {'type': 'str'}})
>>> my_spec.load_config('/path/to/file')
>>> my_spec.load_config({'foo': 'bar'})
>>> my_spec.load_config('ENVIRONMENT')
>>> my_spec.load_config(('label', {'foo': 'bar'}))
>>> my_spec.load_config(('label', '/path/to/file.yaml', 'yaml'))
>>> my_spec.load_config(('label', '/path/to/file.json', 'json'))
```

You can of course combine each of these and the order will be held correctly.

Parameters

- ***args** –
- ****kwargs** – The only supported keyword argument is 'bootstrap' which will indicate that only bootstrap configurations should be loaded.

Returns

A Box object which is subclassed from dict. It should behave exactly as a dictionary. This object is guaranteed to contain at least all of your required configuration items.

Return type box.Box

Raises

- `YapconfLoadError` – If we attempt to load your args and something goes wrong.
- `YapconfItemNotFound` – If an item is required but could not be found in the configuration.
- `YapconfItemError` – If a possible value was found but the type cannot be determined.
- `YapconfValueError` – If a possible value is found but during conversion, an exception was raised.

migrate_config_file(*config_file_path*, *always_update=False*, *current_file_type=None*, *output_file_name=None*, *output_file_type=None*, *create=True*, *update_defaults=True*)

Migrates a configuration file.

This is used to help you update your configurations throughout the lifetime of your application. It is probably best explained through example.

Examples

Assume we have a JSON config file ('/path/to/config.json') like the following: `{"db_name": "test_db_name", "db_host": "1.2.3.4"}`

```
>>> spec = YapconfSpec({
...     'db_name': {
...         'type': 'str',
...         'default': 'new_default',
...         'previous_defaults': ['test_db_name']
...     },
...     'db_host': {
...         'type': 'str',
...         'previous_defaults': ['localhost']
...     }
... })
```

We can migrate that file quite easily with the spec object:

```
>>> spec.migrate_config_file('/path/to/config.json')
```

Will result in /path/to/config.json being overwritten: `{"db_name": "new_default", "db_host": "1.2.3.4"}`

Parameters

- **config_file_path** (*str*) – The path to your current config
- **always_update** (*bool*) – Always update values (even to None)
- **current_file_type** (*str*) – Defaults to self._file_type
- **output_file_name** (*str*) – Defaults to the current_file_path
- **output_file_type** (*str*) – Defaults to self._file_type
- **create** (*bool*) – Create the file if it doesn't exist (otherwise error if the file does not exist).
- **update_defaults** (*bool*) – Update values that have a value set to something listed in the previous_defaults

Returns The newly migrated configuration.

Return type box.Box

update_defaults(*new_defaults*, *respect_none=False*)
Update items defaults to the values in the new_defaults dict.

Parameters

- **new_defaults** (*dict*) – A key-value pair of new defaults to be applied.
- **respect_none** (*bool*) – Flag to indicate if `None` values should constitute an update to the default.

4.1.6 Module contents

Top-level package for Yapconf.

class yapconf.**YapconfSpec**(*specification*, *file_type='json'*, *env_prefix=None*, *encoding='utf-8'*, *separator='.'*)
Bases: `object`

Object which holds your configuration's specification.

The YapconfSpec item is the main interface into the yapconf package. It will help you load, migrate, update and add arguments for your application.

Examples

```
>>> from yapconf import YapconfSpec
```

First define a specification

```
>>> my_spec = YapconfSpec(
... {"foo": {"type": "str", "default": "bar"}},
... env_prefix='MY_APP_')
```

Then load the configuration in whatever order you want! load_config will automatically look for the 'foo' value in '/path/to/config.yml', then the environment, finally falling back to the default if it was not found elsewhere

```
>>> config = my_spec.load_config('/path/to/config.yml', 'ENVIRONMENT')
>>> print(config.foo)
>>> print(config['foo'])
```

add_arguments(*parser*, *bootstrap=False*)
Adds all items to the parser passed in.

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to add all items to.
- **bootstrap** (*bool*) – Flag to indicate whether you only want to mark bootstrapped items as required on the command-line.

defaults
dict – All defaults for items in the specification.

get_item(*name*, *bootstrap=False*)
Get a particular item in the specification.

Parameters

- **name** (*str*) – The name of the item to retrieve.
- **bootstrap** (*bool*) – Only search bootstrap items

Returns (YapconfItem): A YapconfItem if it is found, None otherwise.

load_config(**args*, ***kwargs*)

Load a config based on the arguments passed in.

The order of arguments passed in as *args is significant. It indicates the order of precedence used to load configuration values. Each argument can be a string, dictionary or a tuple. There is a special case string called 'ENVIRONMENT', otherwise it will attempt to load the filename passed in as a string.

By default, if a string is provided, it will attempt to load the file based on the file_type passed in on initialization. If you want to load a mixture of json and yaml files, you can specify them as the 3rd part of a tuple.

Examples

You can load configurations in any of the following ways:

```
>>> my_spec = YapconfSpec({'foo': {'type': 'str'}})
>>> my_spec.load_config('/path/to/file')
>>> my_spec.load_config({'foo': 'bar'})
>>> my_spec.load_config('ENVIRONMENT')
>>> my_spec.load_config(('label', {'foo': 'bar'}))
>>> my_spec.load_config(('label', '/path/to/file.yaml', 'yaml'))
>>> my_spec.load_config(('label', '/path/to/file.json', 'json'))
```

You can of course combine each of these and the order will be held correctly.

Parameters

- ***args** –
- ****kwargs** – The only supported keyword argument is 'bootstrap' which will indicate that only bootstrap configurations should be loaded.

Returns

A Box object which is subclassed from dict. It should behave exactly as a dictionary. This object is guaranteed to contain at least all of your required configuration items.

Return type box.Box

Raises

- YapconfLoadError – If we attempt to load your args and something goes wrong.
- YapconfItemNotFound – If an item is required but could not be found in the configuration.
- YapconfItemError – If a possible value was found but the type cannot be determined.
- YapconfValueError – If a possible value is found but during conversion, an exception was raised.

migrate_config_file(*config_file_path*, *always_update=False*, *current_file_type=None*, *output_file_name=None*, *output_file_type=None*, *create=True*, *update_defaults=True*)

Migrates a configuration file.

This is used to help you update your configurations throughout the lifetime of your application. It is probably best explained through example.

Examples

Assume we have a JSON config file ('/path/to/config.json') like the following: `{"db_name": "test_db_name", "db_host": "1.2.3.4"}`

```
>>> spec = YapconfSpec({
...     'db_name': {
...         'type': 'str',
...         'default': 'new_default',
...         'previous_defaults': ['test_db_name']
...     },
...     'db_host': {
...         'type': 'str',
...         'previous_defaults': ['localhost']
...     }
... })
```

We can migrate that file quite easily with the spec object:

```
>>> spec.migrate_config_file('/path/to/config.json')
```

Will result in /path/to/config.json being overwritten: `{"db_name": "new_default", "db_host": "1.2.3.4"}`

Parameters

- **config_file_path** (*str*) – The path to your current config
- **always_update** (*bool*) – Always update values (even to None)
- **current_file_type** (*str*) – Defaults to self._file_type
- **output_file_name** (*str*) – Defaults to the current_file_path
- **output_file_type** (*str*) – Defaults to self._file_type
- **create** (*bool*) – Create the file if it doesn't exist (otherwise error if the file does not exist).
- **update_defaults** (*bool*) – Update values that have a value set to something listed in the previous_defaults

Returns The newly migrated configuration.

Return type box.Box

update_defaults(*new_defaults*, *respect_none=False*)

Update items defaults to the values in the new_defaults dict.

Parameters

- **new_defaults** (*dict*) – A key-value pair of new defaults to be applied.

- **respect_none** (*bool*) – Flag to indicate if `None` values should constitute an update to the default.

CHAPTER 5

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/loganasherjones/yapconf/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

Yapconf could always use more documentation, whether as part of the official Yapconf 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/loganasherjones/yapconf/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 *yapconf* for local development.

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

```
$ git clone git@github.com:your_name_here/yapconf.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 yapconf
$ cd yapconf/
$ 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 yapconf tests
$ python setup.py test or py.test
$ 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 2.7, 3.4, 3.5, and 3.6. Check https://travis-ci.org/loganasherjones/yapconf/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_yapconf
```

CHAPTER 6

Credits

6.1 Development Lead

- Logan Asher Jones <loganasherjones@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.2.3 (2018-04-03)

- Fixed Python2 unicode error (#41)

0.2.1 (2018-03-11) 0.2.2 (2018-03-28) ———————— * Fixed Python2 compatibility error (#35)

7.2 0.2.1 (2018-03-11)

- Added item to YapconfItemNotFound (#21)
- Removed pytest-runner from setup_reuires (#22)

7.3 0.2.0 (2018-03-11)

- Added auto kebab-case for CLI arguments (#7)
- Added the flag to apply environment prefixes (#11)
- Added `choices` to item specification (#14)
- Added `alt_env_names` to item specification (#13)

7.4 0.1.1 (2018-02-08)

- Fixed bug where `None` was a respected value.

7.5 0.1.0 (2018-02-01)

- First release on PyPI.

Python Module Index

y

Index

N

P

R

S

T

U

Y