# Contents

1 What It Does 3

2 Installation 5

3 Using Confuse 7

4 Credits 9

4.1 Confuse: Painless Configuration 9
   4.1.1 Basic Usage 9
   4.1.2 View Theory 10
   4.1.3 Validation 10
   4.1.4 Command-Line Options 11
   4.1.5 Environment Variables 12
   4.1.6 Search Paths 13
   4.1.7 Manually Specifying Config Files 13
   4.1.8 Your Application Directory 14
   4.1.9 Dynamic Updates 14
   4.1.10 YAML Tweaks 15
   4.1.11 Configuring Large Programs 15
   4.1.12 Redaction 16

4.2 Template Examples 16
   4.2.1 Sequence 16
   4.2.2 MappingValues 17
   4.2.3 Filename 19
   4.2.4 Path 22
   4.2.5 Optional 23

4.3 Changelog 26
   4.3.1 v1.7.0 26
   4.3.2 v1.6.0 26
   4.3.3 v1.5.0 26
   4.3.4 v1.4.0 26
   4.3.5 v1.3.0 26
   4.3.6 v1.2.0 26
   4.3.7 v1.1.0 27
   4.3.8 v1.0.0 27

4.4 API Documentation 27
   4.4.1 Core 27
4.4.2 Exceptions ......................................................... 31
4.4.3 Sources .......................................................... 32
4.4.4 Templates .......................................................... 32
4.4.5 Utility .............................................................. 36
4.4.6 YAML Utility ...................................................... 37

Python Module Index 39

Index 41
Confuse is a configuration library for Python that uses YAML. It takes care of defaults, overrides, type checking, command-line integration, environment variable support, human-readable errors, and standard OS-specific locations.
Here’s what Confuse brings to the table:

- **An utterly sensible API** resembling dictionary-and-list structures but providing **transparent validation** without lots of boilerplate code. Type `config['num_goats'].get(int)` to get the configured number of goats and ensure that it’s an integer.

- Combine configuration data from **multiple sources**. Using **layering**, Confuse allows user-specific configuration to seamlessly override system-wide configuration, which in turn overrides built-in defaults. An in-package `config_default.yaml` can be used to provide bottom-layer defaults using the same syntax that users will see. A runtime overlay allows the program to programmatically override and add configuration values.

- **Look for configuration files in platform-specific paths.** Like `$XDG_CONFIG_HOME` or `~/.config` on Unix; “Application Support” on macOS; `%APPDATA%` on Windows. Your program gets its own directory, which you can use to store additional data. You can transparently create this directory on demand if, for example, you need to initialize the configuration file on first run. And an environment variable can be used to override the directory’s location.

- **Integration with command-line arguments** via `argparse` or `optparse` from the standard library. Use argparse’s declarative API to allow command-line options to override configured defaults.

- **Include configuration values from environment variables.** Values undergo automatic type conversion, and nested dicts and lists are supported.
Confuse is available on PyPI and can be installed using pip:

```
pip install confuse
```
Confuse’s documentation describes its API in detail.
Confuse was made to power beets. Like beets, it is available under the MIT license.

### 4.1 Confuse: Painless Configuration

Confuse is a straightforward, full-featured configuration system for Python.

#### 4.1.1 Basic Usage

Set up your Configuration object, which provides unified access to all of your application’s config settings:

```
config = confuse.Configuration('MyGreatApp', __name__)
```

The first parameter is required; it’s the name of your application, which will be used to search the system for a config file named `config.yaml`. See Search Paths for the specific locations searched.

The second parameter is optional: it’s the name of a module that will guide the search for a defaults file. Use this if you want to include a `config_default.yaml` file inside your package. (The included example package does exactly this.)

Now, you can access your configuration data as if it were a simple structure consisting of nested dicts and lists—except that you need to call the method `.get()` on the leaf of this tree to get the result as a value:

```
value = config['foo'][2]['bar'].get()
```

Under the hood, accessing items in your configuration tree builds up a view into your app’s configuration. Then, `get()` flattens this view into a value, performing a search through each configuration data source to find an answer. (More on views later.)

If you know that a configuration value should have a specific type, just pass that type to `get()`:

```
int_value = config['number_of_goats'].get(int)
```
This way, Confuse will either give you an integer or raise a ConfigTypeError if the user has messed up the configuration. You’re safe to assume after this call that int_value has the right type. If the key doesn’t exist in any configuration file, Confuse will raise a NotFoundError. Together, catching these exceptions (both subclasses of confuse.ConfigError) lets you painlessly validate the user’s configuration as you go.

### 4.1.2 View Theory

The Confuse API is based on the concept of views. You can think of a view as a place to look in a config file: for example, one view might say “get the value for key number_of_goats”. Another might say “get the value at index 8 inside the sequence for key animal_counts”. To get the value for a given view, you resolve it by calling the get() method.

This concept separates the specification of a location from the mechanism for retrieving data from a location. (In this sense, it’s a little like XPath: you specify a path to data you want and then you retrieve it.)

Using views, you can write config['animal_counts'][8] and know that no exceptions will be raised until you call get(), even if the animal_counts key does not exist. More importantly, it lets you write a single expression to search many different data sources without preemptively merging all sources together into a single data structure.

Views also solve an important problem with overriding collections. Imagine, for example, that you have a dictionary called deliciousness in your config file that maps food names to tastiness ratings. If the default configuration gives carrots a rating of 8 and the user’s config rates them a 10, then clearly config['deliciousness']['carrots'].get() should return 10. But what if the two data sources have different sets of vegetables? If the user provides a value for broccoli and zucchini but not carrots, should carrots have a default deliciousness value of 8 or should Confuse just throw an exception? With Confuse’s views, the application gets to decide.

The above expression, config['deliciousness']['carrots'].get(), returns 8 (falling back on the default). However, you can also write config['deliciousness'].get(). This expression will cause the entire user-specified mapping to override the default one, providing a dict object like {'broccoli': 7, 'zucchini': 9}. As a rule, then, resolve a view at the same granularity you want config files to override each other.

**Warning:** It may appear that calling config.get() would retrieve the entire configuration at once. However, this will return only the highest-priority configuration source, masking any lower-priority values for keys that are not present in the top source. This pitfall is especially likely when using Command-Line Options or Environment Variables, which may place an empty configuration at the top of the stack. A subsequent call to config.get() might then return no configuration at all.

### 4.1.3 Validation

We saw above that you can easily assert that a configuration value has a certain type by passing that type to get(). But sometimes you need to do more than just type checking. For this reason, Confuse provides a few methods on views that perform fancier validation or even conversion:

- **as_filename()**: Normalize a filename, substituting tildes and absolute-ifying relative paths. For filenames defined in a config file, by default the filename is relative to the application’s config directory (Configuration.config_dir(), as described below). However, if the config file was loaded with the base_for_paths parameter set to True (see Manually Specifying Config Files), then a relative path refers to the directory containing the config file. A relative path from any other source (e.g., command-line options) is relative to the working directory. For full control over relative path resolution, use the Filename template directly (see Filename).
• **as_choice(choices)**: Check that a value is one of the provided choices. The argument should be a sequence of possible values. If the sequence is a dict, then this method returns the associated value instead of the key.

• **as_number()**: Raise an exception unless the value is of a numeric type.

• **as_pairs()**: Get a collection as a list of pairs. The collection should be a list of elements that are either pairs (i.e., two-element lists) already or single-entry dicts. This can be helpful because, in YAML, lists of single-element mappings have a simple syntax (- key: value) and, unlike real mappings, preserve order.

• **as_str_seq()**: Given either a string or a list of strings, return a list of strings. A single string is split on whitespace.

• **as_str_expanded()**: Expand any environment variables contained in a string using os.path.expandvars(). For example, config['path'].as_filename() ensures that you get a reasonable filename string from the configuration. And calling config['direction'].as_choice(['up', 'down']) will raise a ConfigValueError unless the direction value is either “up” or “down”.

### 4.1.4 Command-Line Options

Arguments to command-line programs can be seen as just another source for configuration options. Just as options in a user-specific configuration file should override those from a system-wide config, command-line options should take priority over all configuration files.

You can use the argparse and optparse modules from the standard library with Confuse to accomplish this. Just call the `set_args` method on any view and pass in the object returned by the command-line parsing library. Values from the command-line option namespace object will be added to the overlay for the view in question. For example, with argparse:

```python
args = parser.parse_args()
config.set_args(args)
```

Correspondingly, with optparse:

```python
options, args = parser.parse_args()
config.set_args(options)
```

This call will turn all of the command-line options into a top-level source in your configuration. The key associated with each option in the parser will become a key available in your configuration. For example, consider this argparse script:

```python
config = confuse.Configuration('myapp')
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='a parameter')
args = parser.parse_args()
config.set_args(args)
print(config['foo'].get())
```

This will allow the user to override the configured value for key `foo` by passing `--foo <something>` on the command line.

Overriding nested values can be accomplished by passing `dots=True` and have dot-delimited properties on the incoming object.

```python
parser.add_argument('--bar', help='nested parameter', dest='foo.bar')
args = parser.parse_args()  # args looks like: {'foo.bar': 'value'}
```

(continues on next page)
set_args works with generic dictionaries too.

```python
args = {
    'foo': {
        'bar': 1
    }
}
config.set_args(args, dots=True)
print(config['foo']['bar'].get())
```

Note that, while you can use the full power of your favorite command-line parsing library, you’ll probably want to avoid specifying defaults in your argparse or optparse setup. This way, Confuse can use other configuration sources—possibly your `config_default.yaml`—to fill in values for unspecified command-line switches. Otherwise, the argparse/optparse default value will hide options configured elsewhere.

### 4.1.5 Environment Variables

Confuse supports using environment variables as another source to provide an additional layer of configuration. The environment variables to include are identified by a prefix, which defaults to the uppercased name of your application followed by an underscore. Matching environment variable names are first stripped of this prefix and then lowercased to determine the corresponding configuration option. To load the environment variables for your application using the default prefix, just call `set_env` on your `Configuration` object. Config values from the environment will then be added as an overlay at the highest precedence. For example:

```bash
export MYAPP_FOO=something
```

```python
import confuse
config = confuse.Configuration('myapp', __name__)
config.set_env()
print(config['foo'].get())
```

Nested config values can be overridden by using a separator string in the environment variable name. By default, double underscores are used as the separator for nesting, to avoid clashes with config options that contain single underscores. Note that most shells restrict environment variable names to alphanumeric and underscore characters, so dots are not a valid separator.

```bash
export MYAPP_FOO__BAR=something
```

```python
import confuse
config = confuse.Configuration('myapp', __name__)
config.set_env()
print(config['foo']['bar'].get())
```

Both the prefix and the separator can be customized when using `set_env`. Note that prefix matching is done to the environment variables prior to lowercasing, while the separator is matched after lowercasing.

```bash
export APPFOO_NESTED_BAR=something
```

```python
import confuse
config = confuse.Configuration('myapp', __name__)
```
conf.set_env(prefix='APP', sep='_nested_')
print(config['foo']['bar'].get())

For configurations that include lists, use integers starting from 0 as nested keys to invoke “list conversion.” If any of the sibling nested keys are not integers or the integers are not sequential starting from 0, then conversion will not be performed. Nested lists and combinations of nested dicts and lists are supported.

export MYAPP_FOO__0=first
export MYAPP_FOO__1=second
export MYAPP_FOO__2__BAR__0=nested

import confuse
config = confuse.Configuration('myapp', __name__)
config.set_env()
print(config['foo'].get())  # ['first', 'second', {'bar': ['nested']}]
import confuse
# Instantiates config. Confuse searches for a config_default.yaml
config = confuse.Configuration('MyGreatApp', __name__)
# Add config items from specified file. Relative path values within the
# file are resolved relative to the application’s configuration directory.
config.set_file('subdirectory/default_config.yaml')
# Add config items from a second file. If some items were already defined,
# they will be overwritten (new file precedes the previous ones). With
# `base_for_paths` set to True, relative path values in this file will be
# resolved relative to the config file’s directory (i.e., 'subdirectory').
config.set_file('subdirectory/local_config.yaml', base_for_paths=True)
val = config['foo']['bar'].get(int)

4.1.8 Your Application Directory

Confuse provides a simple helper, `Configuration.config_dir()`, that gives you a directory used to store
your application’s configuration. If a configuration file exists in any of the searched locations, then the highest-priority
directory containing a config file is used. Otherwise, a directory is created for you and returned. So you can always
expect this method to give you a directory that actually exists.

As an example, you may want to migrate a user’s settings to Confuse from an older configuration system such as
`ConfigParser`. Just do something like this:

```python
config_filename = os.path.join(config.config_dir(),
                                confuse.CONFIG_FILENAME)
with open(config_filename, 'w') as f:
    yaml.dump(migrated_config, f)
```

4.1.9 Dynamic Updates

Occasionally, a program will need to modify its configuration while it’s running. For example, an interactive prompt
from the user might cause the program to change a setting for the current execution only. Or the program might need
to add a derived configuration value that the user doesn’t specify.

To facilitate this, Confuse lets you `assign` to view objects using ordinary Python assignment. Assignment will add an
overlay source that precedes all other configuration sources in priority. Here’s an example of programmatically setting
a configuration value based on a `DEBUG` constant:

```python
if DEBUG:
    config['verbosity'] = 100
...
my_logger.setLevel(config['verbosity'].get(int))
```

This example allows the constant to override the default verbosity level, which would otherwise come from a config-
uration file.

Assignment works by creating a new “source” for configuration data at the top of the stack. This new source takes
priority over all other, previously-loaded sources. You can cause this explicitly by calling the `set()` method on any
view. A related method, `add()`, works similarly but instead adds a new lowest-priority source to the bottom of the
stack. This can be used to provide defaults for options that may be overridden by previously-loaded configuration files.
4.1.10 YAML Tweaks

Confuse uses the PyYAML module to parse YAML configuration files. However, it deviates very slightly from the official YAML specification to provide a few niceties suited to human-written configuration files. Those tweaks are:

- All strings are returned as Python Unicode objects.
- YAML maps are parsed as Python OrderedDict objects. This means that you can recover the order that the user wrote down a dictionary.
- Bare strings can begin with the % character. In stock PyYAML, this will throw a parse error.

To produce a YAML string reflecting a configuration, just call `config.dump()`. This does not cleanly round-trip YAML, but it does play some tricks to preserve comments and spacing in the original file.

Custom YAML Loaders

You can also specify your own PyYAML Loader object to parse YAML files. Supply the `loader` parameter to a `Configuration` constructor, like this:

```python
config = confuse.Configuration("name", loader=yaml.Loaded)
```

To imbue a loader with Confuse’s special parser overrides, use its `add_constructors` method:

```python
class MyLoader(yaml.Loader):
    ...
    confuse.Loader.add_constructors(MyLoader)
config = confuse.Configuration("name", loader=MyLoader)
```

4.1.11 Configuring Large Programs

One problem that must be solved by a configuration system is the issue of global configuration for complex applications. In a large program with many components and many config options, it can be unwieldy to explicitly pass configuration values from component to component. You quickly end up with monstrous function signatures with dozens of keyword arguments, decreasing code legibility and testability.

In such systems, one option is to pass a single `Configuration` object through to each component. To avoid even this, however, it’s sometimes appropriate to use a little bit of shared global state. As evil as shared global state usually is, configuration is (in my opinion) one valid use: since configuration is mostly read-only, it’s relatively unlikely to cause the sorts of problems that global values sometimes can. And having a global repository for configuration option can vastly reduce the amount of boilerplate threading-through needed to explicitly pass configuration from call to call.

To use global configuration, consider creating a configuration object in a well-known module (say, the root of a package). But since this object will be initialized at module load time, Confuse provides a `LazyConfig` object that loads your configuration files on demand instead of when the object is constructed. (Doing complicated stuff like parsing YAML at module load time is generally considered a Bad Idea.)

Global state can cause problems for unit testing. To alleviate this, consider adding code to your test fixtures (e.g., `setUp` in the `unitest` module) that clears out the global configuration before each test is run. Something like this:

```python
config.clear()
config.read(user=False)
```

These lines will empty out the current configuration and then re-load the defaults (but not the user’s configuration files). Your tests can then modify the global configuration values without affecting other tests since these modifications will be cleared out before the next test runs.
4.1.12 Redaction

You can also mark certain configuration values as “sensitive” and avoid including them in output. Just set the `redact` flag:

```
config['key'].redact = True
```

Then flatten or dump the configuration like so:

```
config.dump(redact=True)
```

The resulting YAML will contain “key: REDACTED” instead of the original data.

4.2 Template Examples

These examples demonstrate how the confuse templates work to validate configuration values.

4.2.1 Sequence

A `Sequence` template allows validation of a sequence of configuration items that all must match a subtemplate. The items in the sequence can be simple values or more complex objects, as defined by the subtemplate. When the view is defined in multiple sources, the highest priority source will override the entire list of items, rather than appending new items to the list from lower sources. If the view is not defined in any source of the configuration, an empty list will be returned.

As an example of using the `Sequence` template, consider a configuration that includes a list of servers, where each server is required to have a host string and an optional port number that defaults to 80. For this example, an initial configuration file named `servers_example.yaml` has the following contents:

```
servers:
  - host: one.example.com
  - host: two.example.com
    port: 8000
  - host: three.example.com
    port: 8080
```

Validation of this configuration could be performed like this:

```
>>> import confuse
>>> import pprint
>>> source = confuse.YamlSource('servers_example.yaml')
>>> config = confuse.RootView([source])
>>> template = {
...   'servers': confuse.Sequence({
...     'host': str,
...     'port': 80,
...   }),
... }
>>> valid_config = config.get(template)
>>> pprint.pprint(valid_config)
{'servers': [{'host': 'one.example.com', 'port': 80},
             {'host': 'two.example.com', 'port': 8000},
             {'host': 'three.example.com', 'port': 8080}]}
The list of items in the initial configuration can be overridden by setting a higher priority source. Continuing the previous example:

```python
>>> config.set(
...     'servers': [
...         {'host': 'four.example.org'},
...         {'host': 'five.example.org', 'port': 9000},
...     ],
... )
>>> updated_config = config.get(template)
>>> pprint.pprint(updated_config)
{'servers': [{'host': 'four.example.org', 'port': 80},
             {'host': 'five.example.org', 'port': 9000}]}  
```

If the requested view is missing, `Sequence` returns an empty list:

```python
>>> config.clear()
>>> config.get(template)
{'servers': []}
```

However, if an item within the sequence does not match the subtemplate provided to `Sequence`, then an error will be raised:

```python
>>> config.set(
...     'servers': [
...         {'host': 'bad_port.example.net', 'port': 'default'}
...     ]
... )
>>> try:
...     config.get(template)
... except confuse.ConfigError as err:
...     print(err)
... servers#0.port: must be a number
```

---

**Note:** A python list is not the shortcut for defining a `Sequence` template but will instead produce a `OneOf` template. For example, `config.get([str])` is equivalent to `config.get(confuse.OneOf([str]))` and not `config.get(confuse.Sequence(str))`.

### 4.2.2 MappingValues

A `MappingValues` template allows validation of a mapping of configuration items where the keys can be arbitrary but all the values need to match a subtemplate. Use cases include simple user-defined key:value pairs or larger configuration blocks that all follow the same structure, but where the keys naming each block are user-defined. In addition, individual items in the mapping can be overridden and new items can be added by higher priority configuration sources. This is in contrast to the `Sequence` template, in which a higher priority source overrides the entire list of configuration items provided by a lower source.

In the following example, a hypothetical todo list program can be configured with user-defined colors and category labels. Colors are required to be in hex format. For each category, a description is required and a priority level is optional, with a default value of 0. An initial configuration file named `todo_example.yaml` has the following contents:
colors:
  red: '#FF0000'
  green: '#00FF00'
  blue: '#0000FF'
categories:
  default:
    description: Things to do
  high:
    description: These are important
    priority: 50
  low:
    description: Will get to it eventually
    priority: -10

Validation of this configuration could be performed like this:

```python
>>> import confuse
>>> import pprint

>>> source = confuse.YamlSource('todo_example.yaml')

>>> config = confuse.RootView([source])

>>> template = {
...   'colors': confuse.MappingValues(
...     confuse.String(pattern='#[0-9a-fA-F]{6,6}'),
...   ),
...   'categories': confuse.MappingValues({
...     'description': str,
...     'priority': int,
...   }),
... }

>>> valid_config = config.get(template)

>>> pprint.pprint(valid_config)
{'categories': {'default': {'description': 'Things to do', 'priority': 0},
    'high': {'description': 'These are important', 'priority': 50},
    'low': {'description': 'Will get to it eventually',
    'priority': -10}},
'colors': {'blue': '#0000FF', 'green': '#00FF00', 'red': '#FF0000'}}
```

Items in the initial configuration can be overridden and the mapping can be extended by setting a higher priority source. Continuing the previous example:

```python
>>> config.set({
...   'colors': {
...     'green': '#008000',
...     'orange': '#FFA500',
...   },
...   'categories': {
...     'urgent': {
...       'description': 'Must get done now',
...       'priority': 100,
...     },
...     'high': {
...       'description': 'Important, but not urgent',
...       'priority': 20,
...     },
...   },
... })

>>> updated_config = config.get(template)
```

(continues on next page)
Confuse, Release 1.7.0

>>> pprint.pprint(updated_config)
{'categories': {'default': {'description': 'Things to do', 'priority': 0},
    'high': {'description': 'Important, but not urgent',
             'priority': 20},
    'low': {'description': 'Will get to it eventually',
            'priority': -10},
    'urgent': {'description': 'Must get done now',
               'priority': 100}},
'colors': {'blue': '#0000FF',
           'green': '#008000',
           'orange': '#FFA500',
           'red': '#FF0000')}

If the requested view is missing, MappingValues returns an empty dict:

>>> config.clear()
>>> config.get(template)
{'colors': {}, 'categories': {}}

However, if an item within the mapping does not match the subtemplate provided to MappingValues, then an error will be raised:

>>> config.set(
    ...
    'categories': {
    ...
    'no_description': {
    ...
    'priority': 10,
    ...
    },
    ...
    })
>>> try:
    ...
    config.get(template)
    ...
    except confuse.ConfigError as err:
    ...
    print(err)
    ...
categories.no_description.description not found

4.2.3 Filename

A Filename template validates a string as a filename, which is normalized and returned as an absolute, tilde-free path. By default, relative path values that are provided in config files are resolved relative to the application’s configuration directory, as returned by Configuration.config_dir(), while relative paths from command-line options are resolved from the current working directory. However, these default relative path behaviors can be changed using the cwd, relative_to, in_app_dir, or in_source_dir parameters to the Filename template. In addition, relative path resolution for an entire source file can be changed by creating a ConfigSource with the base_for_paths parameter set to True. Setting the behavior at the source-level can be useful when all Filename templates should be relative to the source. The template-level parameters provide more fine-grained control.

While the directory used for resolving relative paths can be controlled, the Filename template should not be used to guarantee that a file is contained within a given directory, because an absolute path may be provided and will not be subject to resolution. In addition, Filename validation only ensures that the filename is a valid path on the platform where the application is running, not that the file or any parent directories exist or could be created.

Note: Running the example below will create the application config directory ~/.config/ExampleApp/ on MacOS and Unix machines or %APPDATA%\ExampleApp\ on Windows machines. The filenames in the sample
output will also be different on your own machine because the paths to the config files and the current working directory will be different.

For this example, we will validate a configuration with filenames that should be resolved as follows:

- **library**: a filename that should always be resolved relative to the application’s config directory
- **media_dir**: a directory that should always be resolved relative to the source config file that provides that value
- **photo_dir** and **video_dir**: subdirectories that should be resolved relative of the value of **media_dir**
- **temp_dir**: a directory that should be resolved relative to /tmp/
- **log**: a filename that follows the default Filename template behavior

The initial user config file will be at ~/.config/ExampleApp/config.yaml, where it will be discovered automatically using the Search Paths, and has the following contents:

```
library: library.db
media_dir: media
photo_dir: my_photos
video_dir: my_videos
temp_dir: example_tmp
log: example.log
```

Validation of this initial user configuration could be performed as follows:

```
>>> import confuse
>>> import pprint

>>> config = confuse.Configuration('ExampleApp', __name__)  # Loads user config
>>> print(config.config_dir())  # Application config directory
/home/user/.config/ExampleApp

>>> template = {
...    'library': confuse.Filename(in_app_dir=True),
...    'media_dir': confuse.Filename(in_source_dir=True),
...    'photo_dir': confuse.Filename(relative_to='media_dir'),
...    'video_dir': confuse.Filename(relative_to='media_dir'),
...    'temp_dir': confuse.Filename(cwd='/tmp'),
...    'log': confuse.Filename(),
...}

>>> valid_config = config.get(template)

>>> pprint.pprint(valid_config)
{'library': '/home/user/.config/ExampleApp/library.db',
 'log': '/home/user/.config/ExampleApp/example.log',
 'media_dir': '/home/user/.config/ExampleApp/media',
 'photo_dir': '/home/user/.config/ExampleApp/media/my_photos',
 'temp_dir': '/tmp/example_tmp',
 'video_dir': '/home/user/.config/ExampleApp/media/my_videos'}
```

Because the user configuration file config.yaml was in the application’s configuration directory of /home/user/.config/ExampleApp/, all of the filenames are below /home/user/.config/ExampleApp/ except for temp_dir, whose template used the cwd parameter. However, if the following YAML file is then loaded from /var/tmp/example/config.yaml as a higher-level source, some of the paths will no longer be relative to the application config directory:

```
library: new_library.db
media_dir: new_media
photo_dir: new_photos
```

(continues on next page)
# video_dir: my_videos  # Not overridden

temp_dir: ./new_example_tmp
log: new_example.log

Continuing the example code from above:

```python
>>> config.set_file('/var/tmp/example/config.yaml')
>>> updated_config = config.get(template)
>>> pprint.pprint(updated_config)
{'library': '/home/user/.config/ExampleApp/new_library.db',
'log': '/home/user/.config/ExampleApp/new_example.log',
'media_dir': '/var/tmp/example/new_media',
'photo_dir': '/var/tmp/example/new_media/new_photos',
'temp_dir': '/tmp/new_example_tmp',
'video_dir': '/var/tmp/example/new_media/my_videos'}
```

Now, the `media_dir` and its subdirectories are relative to the directory containing the new source file, because the `media_dir` template used the `in_source_dir` parameter. However, `log` remains in the application config directory because it uses the default `Filename` template behavior. The base directories for the `library` and `temp_dir` items are also not affected.

If the previous YAML file is instead loaded with the `base_for_paths` parameter set to True, then a default `Filename` template will use that config file’s directory as the base for resolving relative paths:

```python
>>> config.set_file('/var/tmp/example/config.yaml', base_for_paths=True)
>>> updated_config = config.get(template)
>>> pprint.pprint(updated_config)
{'library': '/home/user/.config/ExampleApp/new_library.db',
'log': '/var/tmp/example/new_example.log',
'media_dir': '/var/tmp/example/new_media',
'photo_dir': '/var/tmp/example/new_media/new_photos',
'temp_dir': '/tmp/new_example_tmp',
'video_dir': '/var/tmp/example/new_media/my_videos'}
```

The filename for `log` is now within the directory containing the new source file. However, the directory for the `library` file has not changed since its template uses the `in_app_dir` parameter, which takes precedence over the source’s `base_for_paths` setting. The template-level `cwd` parameter, used with `temp_dir`, also takes precedence over the source setting.

For configuration values set from command-line options, relative paths will be resolved from the current working directory by default, but the `cwd`, `relative_to`, and `in_app_dir` template parameters alter that behavior. Continuing the example code from above, command-line options are mimicked here by splitting a mock command line string and parsing it with `argparse`:

```python
>>> import os
>>> print(os.getcwd())  # Current working directory
/home/user
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--library')
>>> parser.add_argument('--media_dir')
>>> parser.add_argument('--photo_dir')
>>> parser.add_argument('--temp_dir')
>>> parser.add_argument('--log')
>>> cmd_line=('--library cmd_line_library --media_dir cmd_line_media ' +
'--photo_dir cmd_line_photo --temp_dir cmd_line_tmp')
```

(continues on next page)
Now the `log` and `media_dir` paths are relative to the current working directory of `/home/user`, while the `photo_dir` and `video_dir` paths remain relative to the updated `media_dir` path. The `library` and `temp_dir` paths are still resolved as before, because those templates used `in_app_dir` and `cwd`, respectively.

If a configuration value is provided as an absolute path, the path will be normalized but otherwise unchanged. Here is an example of overriding earlier values with absolute paths:

```python
>>> config.set({
...   'library': '~/home_library.db',
...   'media_dir': '/media',
...   'video_dir': '/video_not_under_media',
...   'temp_dir': '/var/./remove_me/..//tmp',
...   'log': '/var/log/example.log',
... })
>>> absolute_config = config.get(template)
>>> pprint.pprint(absolute_config)
{'library': '/home/user/home_library.db',
 'log': '/var/log/example.log',
 'media_dir': '/media',
 'photo_dir': '/media/cmd_line_photo',
 'temp_dir': '/var/tmp',
 'video_dir': '/video_not_under_media'}
```

The paths for `library` and `temp_dir` have been normalized, but are not impacted by their template parameters. Since `photo_dir` was not overridden, the previous relative path value is now being resolved from the new `media_dir` absolute path. However, the `video_dir` was set to an absolute path and is no longer a subdirectory of `media_dir`.

### 4.2.4 Path

A Path template works the same as a Filename template, but returns a `pathlib.Path` object instead of a string. Using the same initial example as above for Filename but with Path templates gives the following:

```python
>>> import confuse
>>> import pprint
>>> config = confuse.Configuration('ExampleApp', __name__)
>>> print(config.config_dir())  # Application config directory
/home/user/.config/ExampleApp
>>> template = {
...   'library': confuse.Path(in_app_dir=True),
...   'media_dir': confuse.Path(in_source_dir=True),
...   'photo_dir': confuse.Path(relative_to='media_dir'),
...   'video_dir': confuse.Path(relative_to='media_dir'),
... }
```
...  'temp_dir': confuse.Path(cwd='/tmp'),
...  'log': confuse.Path(),
... }

>>> valid_config = config.get(template)

>>> pprint.pprint(valid_config)

{'library': PosixPath('/home/user/.config/ExampleApp/library.db'),
'log': PosixPath('/home/user/.config/ExampleApp/example.log'),
'media_dir': PosixPath('/home/user/.config/ExampleApp/media'),
'photo_dir': PosixPath('/home/user/.config/ExampleApp/media/my_photos'),
'temp_dir': PosixPath('/tmp/example_tmp'),
'video_dir': PosixPath('/home/user/.config/ExampleApp/media/my_videos')

4.2.5 Optional

While many templates like Integer and String can be configured to return a default value if the requested view is missing, validation with these templates will fail if the value is left blank in the YAML file or explicitly set to null in YAML (ie, None in python). The Optional template can be used with other templates to allow its subtemplate to accept null as valid and return a default value. The default behavior of Optional allows the requested view to be missing, but this behavior can be changed by passing allow_missing=False, in which case the view must be present but its value can still be null. In all cases, any value other than null will be passed to the subtemplate for validation, and an appropriate ConfigError will be raised if validation fails. Optional can also be used with more complex templates like MappingTemplate to make entire sections of the configuration optional.

Consider a configuration where log can be set to a filename to enable logging to that file or set to null or not included in the configuration to indicate logging to the console. All of the following are valid configurations using the Optional template with Filename as the subtemplate:

>>> import sys
>>> import confuse

>>> def get_log_output(config):
...     output = config['log'].get(confuse.Optional(confuse.Filename()))
...     if output is None:
...         return sys.stderr
...     return output

>>> config = confuse.RootView([])

>>> config.set({'log': '/tmp/log.txt'})  # `log` set to a filename

>>> get_log_output(config)

'/tmp/log.txt'

>>> config.set({'log': None})  # `log` set to None (ie, null in YAML)

>>> get_log_output(config)

<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>

>>> config.clear()  # Clear config so that `log` is missing

>>> get_log_output(config)

<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>

However, validation will still fail with Optional if a value is given that is invalid for the subtemplate:

>>> config.set({'log': True})

>>> try:
...     get_log_output(config)
... except confuse.ConfigError as err:
...     print(err)

log: must be a filename, not bool

4.2. Template Examples 23
And without wrapping the Filename subtemplate in Optional, null values are not valid:

```python
>>> config.set({'log': None})
>>> try:
...    config['log'].get(confuse.Filename())
... except confuse.ConfigError as err:
...    print(err)
...log: must be a filename, not NoneType
```

If a program wants to require an item to be present in the configuration, while still allowing null to be valid, pass `allow_missing=False` when creating the Optional template:

```python
>>> def get_log_output_no_missing(config):
...    output = config['log'].get(confuse.Optional(confuse.Filename(),
...                        allow_missing=False))
...    if output is None:
...        return sys.stderr
...    return output

>>> config.set({'log': None})  # 'log' set to None is still OK...
>>> get_log_output_no_missing(config)
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
>>> config.clear()  # but 'log' missing now raises an error
>>> try:
...    get_log_output_no_missing(config)
... except confuse.ConfigError as err:
...    print(err)
...log not found
```

The default value returned by Optional can be set explicitly by passing a value to its `default` parameter. However, if no explicit default is passed to Optional and the subtemplate has a default value defined, then Optional will return the subtemplate’s default value. For subtemplates that do not define default values, like MappingTemplate, None will be returned as the default unless an explicit default is provided.

In the following example, Optional is used to make an Integer template more lenient, allowing blank values to validate. In addition, the entire extra_config block can be left out without causing validation errors. If we have a file named `optional.yaml` with the following contents:

```yaml
favorite_number:  # No favorite number provided, but that’s OK
# This part of the configuration is optional. Uncomment to include.
# extra_config:
#  fruit: apple
#  number: 10
```

Then the configuration can be validated as follows:

```python
>>> import confuse
...source = confuse.YamlSource('optional.yaml')
>>> config = confuse.RootView([source])
...# The following 'Optional' templates are all equivalent
...config['favorite_number'].get(confuse.Optional(5))
5
>>> config['favorite_number'].get(confuse.Optional(confuse.Integer(5)))
5
>>> config['favorite_number'].get(confuse.Optional(int, default=5))
5
```

(continues on next page)
Without the `Optional` template wrapping the `Integer`, the blank value in the YAML file will cause an error:

```python
>>> try:
...     config['favorite_number'].get(5)
... except confuse.ConfigError as err:
...     print(err)
... favorite_number: must be a number
```

If the `extra_config` for this example configuration is supplied, it must still match the subtemplate. Therefore, this will fail:

```python
>>> config.set({'extra_config': {}})
>>> try:
...     config['extra_config'].get(confuse.Optional(
...         {'fruit': str, 'number': int},
...         default={}
...     )
... )
... except confuse.ConfigError as err:
...     print(err)
... extra_config.fruit not found
```

But this override of the example configuration will validate:

```python
>>> config.set({'extra_config': {'fruit': 'banana', 'number': 1}})
>>> config['extra_config'].get(confuse.Optional(
...     {'fruit': str, 'number': int},
... )
{ 'fruit': 'banana', 'number': 1 }
```
4.3 Changelog

4.3.1 v1.7.0

- Add support for reading configuration values from environment variables (see EnvSource).
- Resolve a possible race condition when creating configuration directories.

4.3.2 v1.6.0

- A new Configuration.reload method makes it convenient to reload and re-parse all YAML files from the file system.

4.3.3 v1.5.0

- A new MappingValues template behaves like Sequence but for mappings with arbitrary keys.
- A new Optional template allows other templates to be null.
- Filename templates now have an option to resolve relative to a specific directory. Also, configuration sources now have a corresponding global option to resolve relative to the base configuration directory instead of the location of the specific configuration file.
- There is a better error message for Sequence templates when the data from the configuration is not a sequence.

4.3.4 v1.4.0

- pathlib.PurePath objects can now be converted to Path templates.
- AttrDict now properly supports (over)writing attributes via dot notation.

4.3.5 v1.3.0

- Break up the confuse module into a package. (All names should still be importable from confuse.)
- When using None as a template, the result is a value whose default is None. Previously, this was equivalent to leaving the key off entirely, i.e., a template with no default. To get the same effect now, use confuse.REQUIRED in the template.

4.3.6 v1.2.0

- float values (like 4.2) can now be used in templates (just like 42 works as an int template).
- The Filename and Path templates now correctly accept default values.
- It’s now possible to provide custom PyYAML Loader objects for parsing config files.
4.3.7 v1.1.0

- A new `Path` template produces a `pathlib` Path object.
- Drop support for Python 3.4 (following in the footsteps of PyYAML).
- String templates support environment variable expansion.

4.3.8 v1.0.0

The first stable release, and the first that `beets` depends on externally.

4.4 API Documentation

This part of the documentation covers the interfaces used to develop with `confuse`.

4.4.1 Core

Worry-free YAML configuration files.

```python
class confuse.core.ConfigView
    Bases: object

    A configuration “view” is a query into a program’s configuration data. A view represents a hypothetical location in the configuration tree; to extract the data from the location, a client typically calls the `view.get()` method. The client can access children in the tree (subviews) by subscripting the parent view (i.e., `view[key]`).

    add(value)
        Set the `default` value for this configuration view. The specified value is added as the lowest-priority configuration data source.

    all_contents()
        Iterates over all subviews from collections at this view from all sources. If the object for this view in any source is not iterable, then a `ConfigTypeError` is raised. This method is intended to be used when the view indicates a list; this method will concatenate the contents of the list from all sources.

    as_choice(choices)
        Get the value from a list of choices. Equivalent to `get(Choice(choices))`.

    as_filename()
        Get the value as a path. Equivalent to `get(Filename())`.

    as_number()
        Get the value as any number type: int or float. Equivalent to `get(Number())`.

    as_pairs(default_value=None)
        Get the value as a sequence of pairs of two strings. Equivalent to `get(Pairs(default_value=default_value))`.

    as_path()
        Get the value as a `pathlib.Path` object. Equivalent to `get(Path())`.

    as_str()
        Get the value as a (Unicode) string. Equivalent to `get(unicode)` on Python 2 and `get(str)` on Python 3.

    as_str_expanded()
        Get the value as a (Unicode) string, with env vars expanded by `os.path.expandvars()`.
```
as_str_seq(split=True)
Get the value as a sequence of strings. Equivalent to get(StrSeq(split=split)).

exists()
Determine whether the view has a setting in any source.

first()
Return a (value, source) pair for the first object found for this view. This amounts to the first element returned by resolve. If no values are available, a NotFoundError is raised.

flatten(redact=False)
Create a hierarchy of OrderedDicts containing the data from this view, recursively reifying all views to get their represented values.

If redact is set, then sensitive values are replaced with the string “REDACTED”.

get(\textit{template}=<\texttt{object object}>)
Retrieve the value for this view according to the template.

The \textit{template} against which the values are checked can be anything convertible to a \texttt{Template} using \texttt{as_template}. This means you can pass in a default integer or string value, for example, or a type to just check that something matches the type you expect.

May raise a \texttt{ConfigValueError} (or its subclass, \texttt{ConfigTypeError}) or a \texttt{NotFoundError} when the configuration doesn’t satisfy the template.

get_redactions()
Get the set of currently-redacted sub-key-paths at this view.

items()
Iterates over (key, subview) pairs contained in dictionaries from all sources at this view. If the object for this view in any source is not a dict, then a \texttt{ConfigTypeError} is raised.

keys()
Returns a list containing all the keys available as subviews of the current views. This enumerates all the keys in all dictionaries matching the current view, in contrast to \texttt{view.get(dict).keys()}, which gets all the keys for the first dict matching the view. If the object for this view in any source is not a dict, then a \texttt{ConfigTypeError} is raised. The keys are ordered according to how they appear in each source.

name = None
The name of the view, depicting the path taken through the configuration in Python-like syntax (e.g., \texttt{foo['bar'][42]}).

redact
Whether the view contains sensitive information and should be redacted from output.

resolve()
The core (internal) data retrieval method. Generates (value, source) pairs for each source that contains a value for this view. May raise \texttt{ConfigTypeError} if a type error occurs while traversing a source.

root()
The RootView object from which this view is descended.

sequence()
Iterates over the subviews contained in lists from the first source at this view. If the object for this view in the first source is not a list or tuple, then a \texttt{ConfigTypeError} is raised.

set(\textit{value})
\textit{Override} the value for this configuration view. The specified value is added as the highest-priority configuration data source.
**set_args** *(namespace, dots=False)*

Overlay parsed command-line arguments, generated by a library like argparse or optparse, onto this view's value.

**Parameters**

- **namespace** *(dict or Namespace)* – Dictionary or Namespace to overlay this config with. Supports nested Dictionaries and Namespaces.
- **dots** *(bool)* – If True, any properties on namespace that contain dots (\.) will be broken down into child dictionaries. Example:

  ```
  {'foo.bar': 'car'} # Will be turned into {'foo': {'bar': 'car'}}
  ```

**set_redaction**(path, flag)

Add or remove a redaction for a key path, which should be an iterable of keys.

**values()**

Iterates over all the subviews contained in dictionaries from all sources at this view. If the object for this view in any source is not a dict, then a **ConfigTypeError** is raised.

**class confuse.core.Configuration**(appname, modname=None, read=True, loader=<class 'confuse.yaml_util.Loader'>)

**Bases:** confuse.core.RootView

**_add_default_source()**

Add the package’s default configuration settings. This looks for a YAML file located inside the package for the module modname if it was given.

**_add_user_source()**

Add the configuration options from the YAML file in the user’s configuration directory (given by config_dir) if it exists.

**config_dir()**

Get the path to the user configuration directory. The directory is guaranteed to exist as a postcondition (one may be created if none exist).

If the application’s ...DIR environment variable is set, it is used as the configuration directory. Otherwise, platform-specific standard configuration locations are searched for a config.yaml file. If no configuration file is found, a fallback path is used.

**dump**(full=True, redact=False)

Dump the Configuration object to a YAML file.

The order of the keys is determined from the default configuration file. All keys not in the default configuration will be appended to the end of the file.

**Parameters**

- **full** – Dump settings that don’t differ from the defaults as well
- **redact** – Remove sensitive information (views with the redact flag set) from the output

**read**(user=True, defaults=True)

Find and read the files for this configuration and set them as the sources for this configuration. To disable either discovered user configuration files or the in-package defaults, set user or defaults to False.

**reload()**

Reload all sources from the file system.

This only affects sources that come from files (i.e., YamlSource objects); other sources, such as dictionaries inserted with **add** or **set**, will remain unchanged.
set_env(prefix=None, sep='__')
Create a configuration overlay at the highest priority from environment variables.

After prefix matching and removal, environment variable names will be converted to lowercase for use as keys within the configuration. If there are nested keys, list-like dicts (ie, {0: ‘a’, 1: ‘b’}) will be converted into corresponding lists (ie, [‘a’, ‘b’]). The values of all environment variables will be parsed as YAML scalars using the self.loader Loader class to ensure type conversion is consistent with YAML file sources. Use the EnvSource class directly to load environment variables using non-default behavior and to enable full YAML parsing of values.

Parameters

- prefix – The prefix to identify the environment variables to use. Defaults to uppercased self.appname followed by an underscore.
- sep – Separator within variable names to define nested keys.

set_file(filename, base_for_paths=False)
Parses the file as YAML and inserts it into the configuration sources with highest priority.

Parameters

- filename – Filename of the YAML file to load.
- base_for_paths – Indicates whether the directory containing the YAML file will be used as the base directory for resolving relative path values stored in the YAML file. Otherwise, by default, the directory returned by config_dir() will be used as the base.

user_config_path()
Points to the location of the user configuration.

The file may not exist.

class confuse.core.LazyConfig(appname, modname=None)
Bases: confuse.core.Configuration
A Configuration at reads files on demand when it is first accessed. This is appropriate for using as a global config object at the module level.

add(value)
Set the default value for this configuration view. The specified value is added as the lowest-priority configuration data source.

clear()
Remove all sources from this configuration.

read(user=True, defaults=True)
Find and read the files for this configuration and set them as the sources for this configuration. To disable either discovered user configuration files or the in-package defaults, set user or defaults to False.

resolve()
The core (internal) data retrieval method. Generates (value, source) pairs for each source that contains a value for this view. May raise ConfigTypeError if a type error occurs while traversing a source.

set(value)
Override the value for this configuration view. The specified value is added as the highest-priority configuration data source.

class confuse.core.RootView(sources)
Bases: confuse.core.ConfigView
The base of a view hierarchy. This view keeps track of the sources that may be accessed by subviews.
add(obj)
Set the default value for this configuration view. The specified value is added as the lowest-priority configuration data source.

clear()
Remove all sources (and redactions) from this configuration.

get_redactions()
Get the set of currently-redacted sub-key-paths at this view.

resolve()
The core (internal) data retrieval method. Generates (value, source) pairs for each source that contains a value for this view. May raise ConfigTypeError if a type error occurs while traversing a source.

root()
The RootView object from which this view is descended.

set(value)
Override the value for this configuration view. The specified value is added as the highest-priority configuration data source.

set_redaction(path, flag)
Add or remove a redaction for a key path, which should be an iterable of keys.

class confuse.core.Subview(parent, key)
Bases: confuse.core.ConfigView
A subview accessed via a subscript of a parent view.

add(value)
Set the default value for this configuration view. The specified value is added as the lowest-priority configuration data source.

get_redactions()
Get the set of currently-redacted sub-key-paths at this view.

resolve()
The core (internal) data retrieval method. Generates (value, source) pairs for each source that contains a value for this view. May raise ConfigTypeError if a type error occurs while traversing a source.

root()
The RootView object from which this view is descended.

set(value)
Override the value for this configuration view. The specified value is added as the highest-priority configuration data source.

set_redaction(path, flag)
Add or remove a redaction for a key path, which should be an iterable of keys.

4.4.2 Exceptions

exception confuse.exceptions.ConfigError
Bases: exceptions.Exception
Base class for exceptions raised when querying a configuration.

exception confuse.exceptions.NotFoundError
Bases: confuse.exceptions.ConfigError
A requested value could not be found in the configuration trees.

4.4. API Documentation
exception confuse.exceptions.ConfigValueError
    Bases: confuse.exceptions.ConfigError
    The value in the configuration is illegal.

exception confuse.exceptions.ConfigTypeError
    Bases: confuse.exceptions.ConfigValueError
    The value in the configuration did not match the expected type.

exception confuse.exceptions.ConfigTemplateError
    Bases: confuse.exceptions.ConfigError
    Base class for exceptions raised because of an invalid template.

exception confuse.exceptions.ConfigReadError
    Bases: confuse.exceptions.ConfigError
    A configuration source could not be read.

4.4.3 Sources

class confuse.sources.ConfigSource(value, filename=None, default=False, base_for_paths=False)
    Bases: dict
    A dictionary augmented with metadata about the source of the configuration.

classmethod of(value)
    Given either a dictionary or a ConfigSource object, return a ConfigSource object. This lets a function accept either type of object as an argument.

class confuse.sources.EnvSource(prefix, sep='__', lower=True, handle_lists=True, parse_yaml_docs=False, loader=<class 'confuse.yaml_util.Loader'>)
    Bases: confuse.sources.ConfigSource
    A configuration data source loaded from environment variables.

classmethod _convert_dict_lists(obj)
    Recursively search for dicts where all of the keys are integers from 0 to the length of the dict, and convert them to lists.

load()
    Load configuration data from the environment.

class confuse.sources.YamlSource(filename=None, default=False, base_for_paths=False, optional=False, loader=<class 'confuse.yaml_util.Loader'>)
    Bases: confuse.sources.ConfigSource
    A configuration data source that reads from a YAML file.

load()
    Load YAML data from the source’s filename.

4.4.4 Templates

class confuse.templates.AttrDict
    Bases: dict
    A dict subclass that can be accessed via attributes (dot notation) for convenience.
class confuse.templates.Choice(choices, default=<object object>)

Bases: confuse.templates.Template

A template that permits values from a sequence of choices.

convert(value, view)

Ensure that the value is among the choices (and remap if the choices are a mapping).

class confuse.templates.Filename(default=<object object>, cwd=None, relative_to=None, in_app_dir=False, in_source_dir=False)

Bases: confuse.templates.Template

A template that validates strings as filenames.

Filenames are returned as absolute, tilde-free paths.

Relative paths are relative to the template’s cwd argument when it is specified. Otherwise, if the paths come from a file, they will be relative to the configuration directory (see the config_dir method) by default or to the base directory of the config file if either the source has base_for_paths set to True or the template has in_source_dir set to True. Paths from sources without a file are relative to the current working directory. This helps attain the expected behavior when using command-line options.

value(view, template=None)

Get the value for a ConfigView.

May raise a NotFoundError if the value is missing (and the template requires it) or a ConfigValueError for invalid values.

class confuse.templates.Integer(default=<object object>)

Bases: confuse.templates.Template

An integer configuration value template.

convert(value, view)

Check that the value is an integer. Floats are rounded.

class confuse.templates.MappingTemplate(mapping)

Bases: confuse.templates.Template

A template that uses a dictionary to specify other types for the values for a set of keys and produce a validated AttrDict.

value(view, template=None)

Get a dict with the same keys as the view and the value of each item validated against the subtemplate.

class confuse.templates.MappingValues(subtemplate)

Bases: confuse.templates.Template

A template used to validate mappings of similar items, based on a given subtemplate applied to the values.

All keys in the mapping are considered valid, but values must pass validation by the subtemplate. Similar to the Sequence template but for mappings.

value(view, template=None)

Get a dict with the same keys as the view and the value of each item validated against the subtemplate.

class confuse.templates.Number(default=<object object>)

Bases: confuse.templates.Template

A numeric type: either an integer or a floating-point number.

convert(value, view)

Check that the value is an int or a float.
class confuse.templates.OneOf(
    allowed, default=<object object>)
Bases: confuse.templates.Template
A template that permits values complying to one of the given templates.

convert (value, view)
    Ensure that the value follows at least one template.

value (view, template)
    Get the value for a ConfigView.
    May raise a NotFoundError if the value is missing (and the template requires it) or a ConfigValueError for invalid values.

class confuse.templates.Optional (subtemplate, default=None, allow_missing=True)
Bases: confuse.templates.Template
A template that makes a subtemplate optional.
If the value is present and not null, it must validate against the subtemplate. However, if the value is null or missing, the template will still validate, returning a default value. If allow_missing is False, the template will not allow missing values while still permitting null.

value (view, template=None)
    Get the value for a ConfigView.
    May raise a NotFoundError if the value is missing (and the template requires it) or a ConfigValueError for invalid values.

class confuse.templates.Pairs (default_value=None)
Bases: confuse.templates.StrSeq
A template for ordered key-value pairs.
This can either be given with the same syntax as for StrSeq (i.e. without values), or as a list of strings and/or single-element mappings such as:

- key: value
- [key, value]
- key

The result is a list of two-element tuples. If no value is provided, the default_value will be returned as the second element.

class confuse.templates.Path (default=<object object>, cwd=None, relative_to=None, in_app_dir=False, in_source_dir=False)
Bases: confuse.templates.Filename
A template that validates strings as pathlib.Path objects.
Filenames are parsed equivalent to the Filename template and then converted to pathlib.Path objects.
For Python 2 it returns the original path as returned by the Filename template.

value (view, template=None)
    Get the value for a ConfigView.
    May raise a NotFoundError if the value is missing (and the template requires it) or a ConfigValueError for invalid values.

confuse.templates.REQUIRED = <object object>
A sentinel indicating that there is no default value and an exception should be raised when the value is missing.
class confuse.templates.Sequence(subtemplate)
   Bases: confuse.templates.Template

   A template used to validate lists of similar items, based on a given subtemplate.

   value(view, template=None)
       Get a list of items validated against the template.

class confuse.templates.StrSeq(split=True, default=<object object>)
   Bases: confuse.templates.Template

   A template for values that are lists of strings.

   Validates both actual YAML string lists and single strings. Strings can optionally be split on whitespace.

   convert(value, view)
       Convert the YAML-deserialized value to a value of the desired type.

       Subclasses should override this to provide useful conversions. May raise a ConfigValueError when the configuration is wrong.

class confuse.templates.String(default=<object object>, pattern=None, expand_vars=False)
   Bases: confuse.templates.Template

   A string configuration value template.

   convert(value, view)
       Check that the value is a string and matches the pattern.

class confuse.templates.Template(default=<object object>)
   Bases: object

   A value template for configuration fields.

   The template works like a type and instructs Confuse about how to interpret a deserialized YAML value. This includes type conversions, providing a default value, and validating for errors. For example, a filepath type might expand tildes and check that the file exists.

   convert(value, view)
       Convert the YAML-deserialized value to a value of the desired type.

       Subclasses should override this to provide useful conversions. May raise a ConfigValueError when the configuration is wrong.

   fail(message, view, type_error=False)
       Raise an exception indicating that a value cannot be accepted.

       type_error indicates whether the error is due to a type mismatch rather than a malformed value. In this case, a more specific exception is raised.

   get_default_value(key_name='default')
       Get the default value to return when the value is missing.

       May raise a NotFoundError if the value is required.

   value(view, template=None)
       Get the value for a ConfigView.

       May raise a NotFoundError if the value is missing (and the template requires it) or a ConfigValueError for invalid values.

class confuse.templates.TypeTemplate(typ, default=<object object>)
   Bases: confuse.templates.Template

   A simple template that checks that a value is an instance of a desired Python type.
**convert**(value, view)

Convert the YAML-deserialized value to a value of the desired type.

Subclasses should override this to provide useful conversions. May raise a `ConfigValueError` when the configuration is wrong.

**confuse.templates.as_template**(value)

Convert a simple “shorthand” Python value to a `Template`.

### 4.4.5 Utility

**confuse.util.build_dict**(obj, sep=",", keep_none=False)

Recursively builds a dictionary from an `argparse.Namespace`, `optparse.Values`, or dict object.

Additionally, if `sep` is a non-empty string, the keys will be split by `sep` and expanded into a nested dict. Keys with a `None` value are dropped by default to avoid unsetting options but can be kept by setting `keep_none` to `True`.

**Parameters**

- `obj` (`argparse.Namespace` or `optparse.Values` or dict or *): Namespace, Values, or dict to iterate over. Other values will simply be returned.
- `sep` (`str`): Separator to use for splitting properties/keys of `obj` for expansion into nested dictionaries.
- `keep_none` (`bool`): Whether to keep keys whose value is `None`.

**Returns** A new dictionary or the value passed if `obj` was not a dict, Namespace, or Values.

**Return type** dict or *

**confuse.util.config_dirs**()

Return a platform-specific list of candidates for user configuration directories on the system.

The candidates are in order of priority, from highest to lowest. The last element is the “fallback” location to be used when no higher-priority config file exists.

**confuse.util.find_package_path**(name)

Returns the path to the package containing the named module or None if the path could not be identified (e.g., if `name == "__main__"`).

**confuse.util.iter_first**(sequence)

Get the first element from an iterable or raise a ValueError if the iterator generates no values.

**confuse.util.namespace_to_dict**(obj)

If `obj` is `argparse.Namespace` or `optparse.Values` we’ll return a dict representation of it, else return the original object.

Redefine this method if using other parsers.

**Parameters** obj –

**Returns**

**Return type** dict or *

**confuse.util.xdg_config_dirs**()

Returns a list of paths taken from the XDG_CONFIG_DIRS and XDG_CONFIG_HOME environment variables if they exist.
4.4.6 YAML Utility

```python
class confuse.yaml_util.Dumper(stream, default_style=None, default_flow_style=False, canonical=None, indent=None, width=None, allow_unicode=None, line_break=None, encoding=None, explicit_start=None, explicit_end=None, version=None, tags=None, sort_keys=True)
```

**Bases:** `yaml.dumper.SafeDumper`

A PyYAML Dumper that represents OrderedDicts as ordinary mappings (in order, of course).

**represent_bool(data)**

Represent bool as ‘yes’ or ‘no’ instead of ‘true’ or ‘false’.

**represent_list(data)**

If a list has less than 4 items, represent it in inline style (i.e. comma separated, within square brackets).

**represent_none(data)**

Represent a None value with nothing instead of ‘none’.

```python
class confuse.yaml_util.Loader(stream)
```

**Bases:** `yaml.loader.SafeLoader`

A customized YAML loader. This loader deviates from the official YAML spec in a few convenient ways:

- All strings are as are Unicode objects.
- All maps are OrderedDicts.
- Strings can begin with % without quotation.

**static add_constructors(loader)**

Modify a PyYAML Loader class to add extra constructors for strings and maps. Call this method on a custom Loader class to make it behave like Confuse’s own Loader

```python
confuse.yaml_util.load_yaml(filename, loader=<class 'confuse.yaml_util.Loader'>)
```

Read a YAML document from a file. If the file cannot be read or parsed, a ConfigReadError is raised. loader is the PyYAML Loader class to use to parse the YAML. By default, this is Confuse’s own Loader class, which is like SafeLoader with extra constructors.

```python
confuse.yaml_util.load_yaml_string(yaml_string, name, loader=<class 'confuse.yaml_util.Loader'>)
```

Read a YAML document from a string. If the string cannot be parsed, a ConfigReadError is raised. `yaml_string` is a string to be parsed as a YAML document. name is the name to use in error messages. loader is the PyYAML Loader class to use to parse the YAML. By default, this is Confuse’s own Loader class, which is like SafeLoader with extra constructors.

```python
confuse.yaml_util.parse_as_scalar(value, loader=<class 'confuse.yaml_util.Loader'>)
```

Parse a value as if it were a YAML scalar to perform type conversion that is consistent with YAML documents. value should be a string. Non-string inputs or strings that raise YAML errors will be returned unchanged. loader is the PyYAML Loader class to use for parsing, defaulting to Confuse’s own Loader class.

**Examples with the default Loader:**

- ‘1’ will return 1 as an integer
- ‘1.0’ will return 1 as a float
- ‘true’ will return True
- The empty string ‘’ will return None

```python
confuse.yaml_util.restore_yaml_comments(data, default_data)
```

Scan default_data for comments (we include empty lines in our definition of comments) and place them before
the same keys in data. Only works with comments that are on one or more own lines, i.e. not next to a yaml mapping.
C
confuse.core, 27
confuse.exceptions, 31
confuse.sources, 32
confuse.templates, 32
confuse.util, 36
confuse.yaml_util, 37
Symbols

_add_default_source() (confuse.core.Configuration method), 29
_add_user_source() (confuse.core.Configuration method), 29
_convert_dict_lists() (confuse.sources.EnvSource class method), 32

A
add() (confuse.core.ConfigView method), 27
add() (confuse.core.LazyConfig method), 30
add() (confuse.core.RootView method), 30
add() (confuse.core.Subview method), 31
add_constructors() (confuse.yaml_util.Loader static method), 37
all_contents() (confuse.core.ConfigView method), 27
as_choice() (confuse.core.ConfigView method), 27
as_filename() (confuse.core.ConfigView method), 27
as_number() (confuse.core.ConfigView method), 27
as_pairs() (confuse.core.ConfigView method), 27
as_path() (confuse.core.ConfigView method), 27
as_str() (confuse.core.ConfigView method), 27
as_str_expanded() (confuse.core.ConfigView method), 27
as_str_seq() (confuse.core.ConfigView method), 27
as_template() (in module confuse.templates), 36
AttrDict (class in confuse.templates), 32

B
build_dict() (in module confuse.util), 36

C
Choice (class in confuse.templates), 32
clear() (confuse.core.LazyConfig method), 30
clear() (confuse.core.RootView method), 31
config_dir() (confuse.core.Configuration method), 29
config_dirs() (in module confuse.util), 36
ConfigError, 31
ConfigReadError, 32
ConfigSource (class in confuse.sources), 32
ConfigTemplateError, 32
ConfigTypeError, 32
Configuration (class in confuse.core), 29
ConfigValueError, 31
ConfigView (class in confuse.core), 27
confuse.core (module), 27
confuse.exceptions (module), 31
confuse.sources (module), 32
confuse.templates (module), 32
confuse.util (module), 36
confuse.yaml_util (module), 37
convert() (confuse.templates.Choice method), 33
convert() (confuse.templates.Integer method), 33
convert() (confuse.templates.Number method), 33
convert() (confuse.templates.OneOf method), 34
convert() (confuse.templates.String method), 35
convert() (confuse.templates.StrSeq method), 35
convert() (confuse.templates.Template method), 35
convert() (confuse.templates.TypeTemplate method), 35

D
dump() (confuse.core.Configuration method), 29
Dumper (class in confuse.yaml_util), 37

E
EnvSource (class in confuse.sources), 32
exists() (confuse.core.ConfigView method), 28

F
fail() (confuse.templates.Template method), 35
Filename (class in confuse.templates), 33
find_package_path() (in module confuse.util), 36
first() (confuse.core.ConfigView method), 28
flatten() (confuse.core.ConfigView method), 28
Index
value() (confuse.templates.OneOf method), 34
value() (confuse.templates.Optional method), 34
value() (confuse.templates.Path method), 34
value() (confuse.templates.Sequence method), 35
value() (confuse.templates.Template method), 35
values() (confuse.core.ConfigView method), 29

X
xdg_config_dirs() (in module confuse.util), 36

Y
YamlSource (class in confuse.sources), 32