Confuse

Release 1.3.0

Oct 15, 2020

Contents

Confuse is a configuration library for Python that uses YAML. It takes care of defaults, overrides, type checking, command-line integration, human-readable errors, and standard OS-specific locations.

CHAPTER 1

What It Does

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.

CHAPTER 2

Installation

Confuse is available on PyPI and can be installed using `pip`:

```
pip install confuse
```

CHAPTER 3

Using Confuse

Confuse's documentation describes its API in detail.

CHAPTER 4

Credits

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 preemtively 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 10 (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.

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. The filename is relative to the source that provided it. That is, a relative path in a config file refers to the directory containing the config file. A relative path in the defaults refers to the application's config directory (`config.config_dir()`, as described below). A relative path from any other source (e.g., command-line options) is relative to the working directory.
- `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:

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

Correspondingly, with optparse:

```
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:

```
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.

```
parser.add_argument('--bar', help='nested parameter', dest='foo.bar')
args = parser.parse_args()  # args looks like: {'foo.bar': 'value'}
config.set_args(args, dots=True)
print(config['foo']['bar'].get())
```

set_args works with generic dictionaries too.

```
args = {
  'foo': {
    'bar': 1
  }
}
```

(continues on next page)

(continued from previous page)

```
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 Search Paths

Confuse looks in a number of locations for your application's configurations. The locations are determined by the platform. For each platform, Confuse has a list of directories in which it looks for a directory named after the application. For example, the first search location on Unix-y systems is `$XDG_CONFIG_HOME/AppName` for an application called `AppName`.

Here are the default search paths for each platform:

- macOS: `~/.config/app` and `~/Library/Application Support/app`
- Other Unix: `~/.config/app` and `/etc/app`
- Windows: `%APPDATA%\app` where the *APPDATA* environment variable falls back to `%HOME%\AppData\Roaming` if undefined

Both macOS and other Unix operating sytems also try to use the `XDG_CONFIG_HOME` and `XDG_CONFIG_DIRS` environment variables if set then search those directories as well.

Users can also add an override configuration directory with an environment variable. The variable name is the application name in capitals with "DIR" appended: for an application named `AppName`, the environment variable is `APPNAMEDIR`.

4.1.6 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:

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

4.1.7 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:

```
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 configuration 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.

4.1.8 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:

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

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

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

4.1.9 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 unittest module) that clears out the global configuration before each test is run. Something like this:

```
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.

4.1.10 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.

4.1.11 Changelog

v1.4.0

- *pathlib.PurePath* objects can now be converted to *Path* templates.

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.

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.

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.

v1.0.0

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

4.2 API Documentation

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

4.2.1 Core

Worry-free YAML configuration files.

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]`).

classmethod _build_namespace_dict(*obj*, *dots=False*)
Recursively replaces all argparse.Namespace and optparse.Values with dicts and drops any keys with None values.

Additionally, if dots is True, will expand any dot delimited keys.

Parameters

- **obj** (*argparse.Namespace or optparse.Values or dict or **) – Namespace, Values, or dict to iterate over. Other values will simply be returned.
- **dots** – If True, any properties on obj that contain dots (.) will be broken down into child dictionaries.

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

Return type dict or *

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(*template=<object object>*)
Retrieve the value for this view according to the template.

The *template* against which the values are checked can be anything convertible to a *Template* using *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 *ConfigValueError* (or its subclass, *ConfigTypeError*) or a *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 *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 `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 *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., `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 *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_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*.

set_file(*filename*)
Parses the file as YAML and inserts it into the configuration sources with highest priority.

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.

4.2.2 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.

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**(*filename*, *reason=None*)
Bases: *confuse.exceptions.ConfigError*

A configuration file could not be read.

4.2.3 Sources

class confuse.sources.**ConfigSource**(*value*, *filename=None*, *default=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.**YamlSource**(*filename=None*, *default=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.

4.2.4 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*)
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, then the configuration directory (see the *config_dir* method) if they come from a file. Otherwise, they 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 template and values validated according to the value types.

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.**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*)
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.

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*.

4.2.5 Utility

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 varibables if they exist

4.2.6 YAML Utility

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'.

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 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

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.

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.

Python Module Index

C

Index

Symbols

A

C

D

E

F

G