Confuse

Release 1.7.0

Jul 16, 2022

Contents

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.

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.
- Include configuration values from **environment variables**. Values undergo automatic type conversion, and nested dicts and lists are supported.

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 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:

```
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'}
```

(continues on next page)

(continued from previous page)

```
config.set_args(args, dots=True)
print(config['foo']['bar'].get())
```

set_args works with generic dictionaries too.

```
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:

```
export MYAPP_FOO=something
```

```
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.

```
export MYAPP_FOO__BAR=something
```

```
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.

```
export APPFOO_NESTED_BAR=something
```

```
import confuse
config = confuse.Configuration('myapp', __name__)
```

(continues on next page)

(continued from previous page)

```
config.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']}]
```

For consistency with YAML config files, the values of environment variables are type converted using the same YAML parser used for file-based configs. This means that numeric strings will be converted to integers or floats, "true" and "false" will be converted to booleans, and the empty string or "null" will be converted to `None`. Setting an environment variable to the empty string or "null" allows unsetting a config value from a lower-precedence source.

To change the lowercasing and list handling behaviors when loading environment variables or to enable full YAML parsing of environment variables, you can initialize an `EnvSource` configuration source directly.

If you use config overlays from both command-line args and environment variables, the order of calls to `set_args` and `set_env` will determine the precedence, with the last call having the highest precedence.

4.1.6 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`.

4.1.7 Manually Specifying Config Files

You may want to leverage Confuse's features without *Search Paths*. This can be done by manually specifying the YAML files you want to include, which also allows changing how relative paths in the file will be resolved:

```
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:

```
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:

```
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.

4.1.10 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:

```
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)
```

4.1.11 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 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.

4.1.12 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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': 0,
...     }),
... }
>>> 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:

```
>>> 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)

(continued from previous page)

```
>>> 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)

(continued from previous page)

```
# video_dir: my_videos  # Not overridden
temp_dir: ./new_example_tmp
log: new_example.log
```

Continuing the example code from above:

```
>>> 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:

```
>>> 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`:

```
>>> 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)

(continued from previous page)

```
...             '--log cmd_line_log')
>>> args = parser.parse_args(cmd_line.split())
>>> config.set_args(args)
>>> config_with_cmdline = config.get(template)
>>> pprint.pprint(config_with_cmdline)
{'library': '/home/user/.config/ExampleApp/cmd_line_library',
 'log': '/home/user/cmd_line_log',
 'media_dir': '/home/user/cmd_line_media',
 'photo_dir': '/home/user/cmd_line_media/cmd_line_photo',
 'temp_dir': '/tmp/cmd_line_tmp',
 'video_dir': '/home/user/cmd_line_media/my_videos'}
```

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 overridding earlier values with absolute paths:

```
>>> 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:

```
>>> 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'),
```

(continues on next page)

(continued from previous page)

```
...     '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
```

And without wrapping the `Filename` subtemplate in `Optional`, `null` values are not valid:

```
>>> 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:

```
>>> 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:

```
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:

```
>>> 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)

(continued from previous page)

```
>>> # But a default passed to `Optional` takes precedence and can be any type
... config['favorite_number'].get(confuse.Optional(5, default='five'))
'five'
>>> # `Optional` with `MappingTemplate` returns `None` by default
... extra_config = config['extra_config'].get(confuse.Optional(
...     {'fruit': str, 'number': int},
... ))
>>> print(extra_config is None)
True
>>> # But any default value can be provided, like an empty dict...
... config['extra_config'].get(confuse.Optional(
...     {'fruit': str, 'number': int},
...     default={},
... ))
{}
>>> # or a dict with default values
... config['extra_config'].get(confuse.Optional(
...     {'fruit': str, 'number': int},
...     default={'fruit': 'orange', 'number': 3},
... ))
{'fruit': 'orange', 'number': 3}
```

Without the `Optional` template wrapping the `Integer`, the blank value in the YAML file will cause an error:

```
>>> 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:

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

But this override of the example configuration will validate:

```
>>> 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 v2.0.0

- Drop support for versions of Python below 3.6.

4.3.2 v1.7.0

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

4.3.3 v1.6.0

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

4.3.4 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.

4.3.5 v1.4.0

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

4.3.6 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.

4.3.7 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.

4.3.8 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.

4.3.9 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

4.4.2 Exceptions

4.4.3 Sources

4.4.4 Templates

4.4.5 Utility

4.4.6 YAML Utility