Confuse

Release 0.1.0

July 02, 2016

Contents

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

CHAPTER 1

Using Confuse

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 that will be used to search the system for config files. 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 file, performing a search through each configuration data source to find an answer. More on view 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.

CHAPTER 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.

CHAPTER 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.

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".

CHAPTER 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.

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.

CHAPTER 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:

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

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`.

CHAPTER 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)
```

CHAPTER 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 be 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.

CHAPTER 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 file reflecting a configuration, just call `config.dump()`. If you supply a filename, the YAML will be written to the file; otherwise, a string is returned. This does not cleanly round-trip YAML, but it does play some tricks to preserve comments and spacing in the original file.

CHAPTER 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.

CHAPTER 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.