Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Program Configuration in Python

Despite numerous options for passing config data to a program, there is still a need for a utility to handle complex hierarchical configuration and locate config files on distributed system. Here is one.

November 18, 2014
URL:http://www.drdobbs.com/open-source/program-configuration-in-python/240169310

Computer programs are made of code. However, most nontrivial programs can be configured to behave in different ways without changing the code. There are many ways of configuring a program such as: command-line arguments, environment variables, configuration files, reading configuration information from a database, and reading configuration data over the network. Each form of configuration is appropriate for certain situations. Many programs combine several forms of configuration. In this article, I explore the spectrum of configuration options for single programs, distributed processes (same program running on multiple cores and/or machines), and distributed systems (a collection of different programs running on multiple cores and/or machines). I will also present a Python package that can help with managing configuration when dealing with systems composed of multiple configurable components.

While the sample code is in Python, the concepts apply just the same to any programming language including compiled languages.

Configuration is typically information provided to the program that controls operational aspects at runtime. Some examples are:

- How much memory to use
- The size of the thread pool
- The database URL
- Number of retries when an operation fails
- The delay in seconds between retry attempts
- What port to listen on

There are many configuration mechanisms. Let's consider the pros and cons of each one and how multiple mechanisms can be used together to provide several options to configure the same program.

Command-Line Arguments

Command-line arguments are very flexible. You can launch different instances of the same program with different command-line arguments. Most programming languages also have good modules or libraries for documenting and parsing command-line arguments. In Python, the argparse module is all the rage these days.

However, command-line arguments have disadvantages: You have to enter them each and every time you want to run a program. This can be a tedious and error-prone process, especially if you are dealing with large volumes. Command-line arguments don't excel when the configuration contains spaces and/or quotes and needs to be escaped to avoid mis-parsing. Also, command-line arguments are not great for hierarchical data for obvious reasons. Another problem with command-line arguments for configuration is that the user specifies these arguments, which can lead to a steep learning-curve for non-techies and the consequent errors. Finally, command-line arguments can be passed only once. There is no way to update command-line arguments while the program is running.

Using Launchers with Fixed Command-Line Arguments

If you launch a program with the same command-line arguments over and over again, you may work around the drudgery by defining a launcher (shell script, alias, shortcut, or .bat file on Windows) that encodes the command-line arguments.

For example, if I always want to run sum.py with the same argument, I could write a little shell launcher called sum.sh:

```
python sum.py input.txt 3
```

Now, I can run it anytime just by typing sum.sh (assuming it is executable).

Launchers can also encode just *some* of the command-line arguments and let the user enter additional arguments. This allows sysadmins more control over the operation of the program by providing a launcher that exposes to the user only input arguments, such as the name of the input file, but encodes operational arguments, such as how long to wait after the program is done.

Environment Variables

Environment variables are often used for configuration because you can set them once per user and they will persist for all invocations of the program. Here is how to read from an environment variable in Python:

```
# Get the delay from the environ variable SUM_DELAY and convert to a float
delay = float(os.environ['SUM_DELAY'])
```

There are various ways for sysadmins to control the user environment, and often certain programs are assigned to specific accounts with a managed environment or launchers set the environment variables before invoking the target program.

But environment variables are not a configuration panacea either. They suffer from many of the issues associated with command-line arguments. They are not suitable for large amounts of configuration or hierarchical configuration (although there are no parsing issues, as each environment variable is separate).

A potentially big problem with environment variables involves naming conflicts. Suppose I named the environment variable DELAY, and another program also used an environment variable called DELAY. If both programs run in the same environment, only one environment variable called DELAY can be set and both will share the its value. Thus, a common practice is to prefix the environment variable with a (hopefully) unique prefix.

A final note on environment variable limitations: The environment is a shared resource and polluting it with lots of variables needed by a specific program is just bad form.

Configuration Files

The configuration file is the workhorse of configuration management. It is the preferred method of configuration for lots of programs that have many configuration options or hierarchical configuration. Configuration files come in an assortment of formats.

In the early days of Windows, .ini files were king (very simple, but only one native level of nesting). Then XML came along and it became a popular configuration file format (sadly, still with us on platforms like Android).

The Web 2.0 and Ajax revolution brought JSON as a more popular alternative. On the UNIX side, simple key = value .conf files are very common.

In Python, a common option that blurs the lines between code and configuration is to import a Python file. I will not dwell on it because it is solely a dynamic language trick, but check the configuration documentation for Web frameworks like Django or Flask if you're curious.

Listing One reads the delay from a configuration file. I'll use my favorite YAML format. Here is the config file conf.yaml,: {delay:3}

Listing One

```
import sys
import time
import yaml

# Get the name of the input file
filename = sys.argv[1]

# Get the delay and convert to a float
delay = yaml.load(

# Read the numbers from file
numbers = [int(x) for x in open(filename).read().split()]

# Print the sum
print (sum(numbers))

# Wait the designated number of seconds
time.sleep(delay)
```

Configuration files can provide a lot of flexibility via search paths. For example, a default configuration file for some component `x` may be in some standard location such as `/etc/x/x.conf`. But, if there is a configuration file in the current user's directory `~/.x/x.conf`, then it overrides the configuration in the default `x.conf`; and if a configuration filename is provided via an environment variable or command-line argument (as in the example above), then it takes precedence.

Configuration files are handy, but they have to be managed carefully. There are several issues that can arise with configuration files, especially when dealing with large deployments.

The first issue is, where do you keep them? Some configuration files contain sensitive information such as user names, passwords, API keys, etc. Storing them in plaintext in your version-control system may not be appropriate. Another issue is that different environments may require different configurations, so while there may be a single version of the code running in each environment (dev, staging, production), there may be different configuration files per environment or even per machine. This issue exists with other forms of configuration like command-line arguments and environment variables, too, but the difference is that people often assume that you can just treat configuration files like source code files. Another issue with very modular systems is that there may be many configuration files for different components/modules/libraries. A single program may be composed of multiple components and each one may have its own configuration file. There may be other programs using the same component, but with a different configuration. For example, consider a logging package, where you need to provide a log filename and other information via a configuration file. Multiple programs will need to use this logging component and each one may want to have its own log filename and other parameters (such as minimal log level). In large systems composed of multiple programs that share multiple configurable components, this can lead to a combinatorial explosion of configuration files, which is a nightmare to manage.

Distributed Configuration

Distributed configuration is an approach for managing configuration in a distributed system. It addresses many of the problems associated with using configuration files. The concept is that the configuration data is stored in a centrally available store. You could roll your own using a database or a shared file system, but it is probably in your best interest to use an existing distributed configuration store. A distributed configuration store has an API and/or client libraries you can use from anywhere to get and set configuration information (if you have access). In this article, I'll use etcd as a distributed configuration store.

The benefits are that a single store can be globally maintained and provide a solid foundation to solve hard problems like synchronization, rolling updates,

and versioning. Because the configuration store is a critical component of the system, you have to make sure it is sufficiently available and redundant.

Combining Options

Sometimes configuration uses a combination of multiple mechanisms: some distributed configuration, some configuration files, some environment variables, and some command-line arguments. In reality, every nontrivial enterprise with multiple programs and services running is pretty much guaranteed to be combining options. There are several reasons for this situation:

- Some organizations don't have an explicit policy regarding configuration and every developer or team comes up with their own solution.
- Different mechanisms are appropriate in different situations as discussed earlier.
- Legacy code has staying power. Even if there is a good policy in place it doesn't mean that all existing software should be modified.

Distributed Configuration vs. One Big Configuration File

Some people have a natural aversion to distributed configuration, even for distributed systems. The argument goes like this: We have a distributed system and we can already deploy code reliably to N servers. Why can't we just have one big configuration file per application, and whenever needed, just deploy it to all the N servers the application is running on?

There are several drawbacks to this approach:

- A big configuration update might often partially fail, and you end up with some servers running old configurations and some servers running new configurations.
- Some configuration settings (such as a database URL) are shared by multiple applications. You will need to duplicate these settings in the big configuration file of each application.
- Different environments (dev, staging, production) will differ in some of their settings (for example, that same database URL). You will have to maintain different configuration files for each environment.

If you have three or four environments and tens or hundreds of applications that share different subsets of their configuration, things become very cumbersome.

ConMan

ConMan (short for "configuration manager") is an open-source Python package I wrote that manages and provides an easy-to-use interface for complex hierarchical configurations. The source is available on GitHub. It is designed to help when you have many configuration files in various formats or if you want to use etcd as a distributed configuration store. In both cases, it exposes all the configuration information as a Python dictionary.

ConMan doesn't deal with command-line arguments or environment variables because:

- I wanted to focus on the more complex case of hierarchical configuration data shared by multiple programs. Command-line arguments in particular are designed for a single run of a program, and environment variables are typically used for flat key-value pairs (although the values could be something like an XML or JSON string with internal structure).
- There are excellent tools to deal with command-line arguments (argparse), and environment variables are already exposed as a dictionary (os.environ).

Let's go over ConMan's code to understand what it does and how to use it. Figure 1 shows the directory structure:

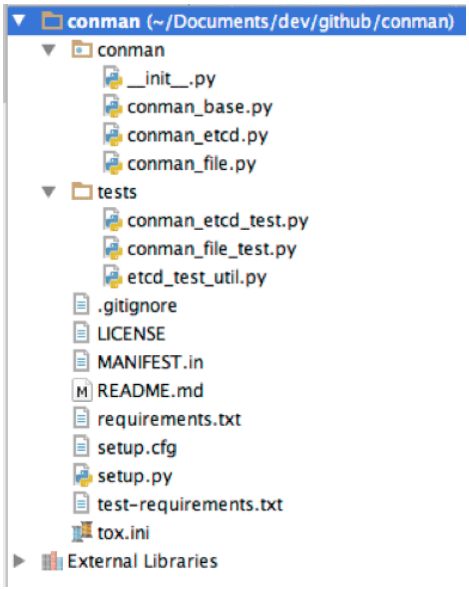

Figure 1: Directory structure.

ConManBase

The conman_base.py file contains the `ConManBase` class. This class serves as a base class to both `ConManFile` and `ConManEtcd`. It provides the `private _conf` attribute that actually stores the configuration, a `__repr__()` method, which is the text representation of Python objects (for example, it's called when "print" is used), and a dictionary interface that forwards get requests to the `internal _conf` and raises an exception if a program attempts to modify the configuration:

```
class ConManBase(dict):
    def __init__(self):
        self._conf = {}

    def __getitem__(self, k):
        return self._conf.__getitem__(k)

    def __setitem__(self, k, v):
        raise NotImplementedError

    def __repr__(self):
        return repr(self._conf)
```

ConManFile

The conman_file.py class `ConManFile` deals with configuration files of several formats. It is a little more complicated.

This section has the necessary imports (including the base class) and the list of supported file formats: .ini, .json, and .yaml. I chose not to support XML because it's such a crusty and not human-friendly format that I stopped using it years ago. But it is still very common (especially in the Java world), so it may be reasonable to add XML support later.

```
import os
import json
import yaml
from ConfigParser import SafeConfigParser
from conman.conman_base import ConManBase

FILE_TYPES = 'ini json yaml'.split()
```

The `ConManFile` class subclasses `ConManBase` to get all its goodies. The `__init__()` method (called when an object is instantiated) accepts a list of configuration filenames. It initializes the base class, initializes an internal list of configuration files, then calls the `add_config_file()` for each of the input config files.

```
class ConManFile(ConManBase):
    def __init__(self, config_files=()):
        ConManBase.__init__(self)
        self._config_files = []
        for config_file in config_files:
            self.add_config_file(config_file)
```

The `add_config_file()` is the only public method of `ConManFile`. It enables the user to dynamically add configuration files in addition to the list of configuration files provided when instantiating a `ConManFile` object. It also allows periodically refreshing existing config files.

There are many ways to tell `add_config_file()` about a configuration file. Here are the rules for the various arguments it accepts:

- The filename contains the path to the config file. The filename may also be read from an environment variable. Either `filename` is not `None` or `environment_Variable` is not `None`, but not both.
- If `base_dir` is not `None`, then it will be combined with the config filename to create an absolute path.
- If a file type is provided, it is used to determine how to parse the config file. If no file type is provided, ConMan will try to guess by the extension using the `_guess_file_type()` method (described later).

If the rules are violated, `add_config_file()` raises an exception. Listing Two show its workings.

Listing Two

```
    def add_config_file(self,
                        filename=None,
                        env_variable=None,
                        base_dir=None,
                        file_type=None):
        if filename is None and env_variable is None:
            raise Exception('filename and env_variable are both None')

        if filename is not None and env_variable is not None:
            raise Exception('filename and env_variable are both not None')

        if filename in self._config_files:
            raise Exception('filename is already in the config file list')

        if env_variable:
            filename = os.environ[env_variable]

        if base_dir:
            filename = os.path.join(base_dir, filename)

        if not os.path.isfile(filename):
            raise Exception('No such file: ' + filename)

        if not file_type:
```

```
            file_type = self._guess_file_type(filename)
```

At this point, we have a filename and possibly a file type, too. If there is a file type, ConMan tries to process the file by calling the `_process_file()` method. If it succeeds, we're done. If it fails or if no file type could be guessed in the first place, then `add_config_file()` simply iterates over all its known file types and tries to process the input file as each one of them until one succeeds. If the input file failed to be processed as any of the file types, a 'Bad config file' exception is raised, as shown in Listing Three.

Listing Three

```
        file_types = set(FILE_TYPES)
        if file_type:
            try:
                return self._process_file(filename, file_type)
            except:
                # Remove failed file_type from set of file types to try
                if file_type in file_types:
                    file_types.remove(file_type)

        # If no file type can be guessed or guess failed try all parsers
        for file_type in file_types:
            try:
                return self._process_file(filename, file_type)
            except:
                pass

        raise Exception('Bad config file: ' + filename)
```

The `_guess_file_type()` just gets the extension of the input filename and compares it to the supported file types (yaml can have two extensions). It returns the file type or `None` if there's no match (Listing Four).

Listing Four

```
    def _guess_file_type(self, filename):
        ext = os.path.splitext(filename)[1][1:]
        return dict(yml='yaml',
                    yaml='yaml',
                    json='json',
                    ini='ini').get(ext, None)
```

The `_process_file()` method is just a delegator. Based on the file type, it constructs the corresponding method name and invokes it. The actual process methods follow a uniform naming scheme of `_process_<file_type>_file()`. The various process methods parse the file and then populate the `_conf` dictionary (defined in the base class), as in Listing Five.

Listing Five

```
    def _process_file(self, filename, file_type):
        process_func = getattr(self, '_process_%s_file' % file_type)
        process_func(filename)

    def _process_ini_file(self, filename):
        parser = SafeConfigParser()
        parser.read(filename)
        for section_name in parser.sections():
            self._conf[section_name] = {}
            section = self._conf[section_name]
            for name, value in parser.items(section_name):
                section[name] = value

    def _process_json_file(self, filename):
        self._conf.update(json.load(open(filename)))

    def _process_yaml_file(self, filename):
        self._conf.update(yaml.load(open(filename)))
```

ConManEtcd

The conman_etcd.py file contains the `ConManEtcd` class, which allows you to add the content of various etcd keys as nested dictionaries, then expose them as one unified dictionary (via the `ConManBase` base class).

The first section is just some necessary imports. The etcd import is the etcd client module.

```
import functools
import etcd
import time
from conman.conman_base import ConManBase
```

etcd is a distributed server, so it often responds with varying latencies or may even be unreachable temporarily (you can control it, of course, with proper networking and redundancy). A `@thrice` decorator is an easy way to interact in a logical and controlled way, but it may exhibit transient failure and require a few tries to get right. The idea is that a callable function or method that is decorated with `@thrice` will automatically be called up to three times. If any of the calls succeed and no exception is raised, it returns immediately with the result. If the first or second calls fail, then it waits a bit to give the service some time to recover and tries again. If the call fails three times, then `@thrice` gives up and just re-raises the exception, which propagates to the caller (Listing Six). The `ConManEtcd` class utilizes it when it connects via the `etcd.Client`.

Listing Six

```
def thrice(delay=0.5):
    def decorated(f):
        @functools.wraps(f)
        def wrapped(*args, **kwargs):
            for i in xrange(3):
                try:
                    return f(*args, **kwargs)
                except Exception:
                    if i == 2:
                        raise
                    time.sleep(delay)
        return wrapped
    return decorated
```

The `ConManEtcd` class exposes two public methods (in addition to dictionary read-only access to the configuration information via the base class). The `add_key()` reads a key (and recursively all its sub-keys and values), and populates the `_conf` dictionary of the base class. The `refresh()` method can refresh a single key or all the keys currently managed. This allows the ability to periodically ensure that the configuration is up-to-date, which is important for long-running systems (Listing Seven).

Listing Seven

```
class ConManEtcd(ConManBase):
    def __init__(self, host='127.0.0.1', port=4001, allow_reconnect=True):
        ConManBase.__init__(self)
        self._connect(host, port, allow_reconnect)

    @thrice()
    def _connect(self, host, port, allow_reconnect):
        self.client = etcd.Client(
            host=host,
            port=port,
            allow_reconnect=allow_reconnect)

    def _add_key_recursively(self, target, key, etcd_result):
        if key.startswith('/'):
            key = key[1:]
        if etcd_result.value:
            target[key] = etcd_result.value
        else:
            target[key] = {}
            target = target[key]
            for c in etcd_result.children:
                k = c.key.split('/')[-1]
                self._add_key_recursively(target, k, c)

    def add_key(self, key):
        etcd_result = self.client.read(key, recursive=True, sorted=True)
        self._add_key_recursively(self._conf, key, etcd_result)

    def refresh(self, key=None):
        keys = [key] if key else self._conf.keys()
        for k in keys:
            self.add_key(k)
```

Using ConMan

To use ConMan in your code, simply install the package and its requirements. Get it from GitHub, then run:

```
pip install conman
```

For usage examples, refer to the tests. In the source code, etcd_test_util.py includes helper functions to manage a local etcd server.

Conclusion

In this article, I discussed the spectrum of configuration options ranging from simple programs up to large distributed systems. I also introduced a Python package designed to help manage multiple configuration files and distributed configuration for more intricate use cases. I hope my package helps with managing configuration on systems composed of multiple configurable components, or inspires ideas for creating you own.

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems.