oscar.flag Documentation

Release 0.1.0

Oscar Engineering

February 08, 2016

Contents

`oscar.flag` provides extensible, namespaced flags which can be parsed from environment variables, command-line arguments and config files.

Flags are declared where they are used in an application or library, and they are accessed through a namespace matching their fully qualified module path.

Documentation lives at Read the Docs, the code on GitHub.

CHAPTER 1

Example

Application entry-point:

```
import sys
from oscar import flag

import other_module


FLAGS = flag.namespace(__name__)
FLAGS.some_int = flag.Int('some integer value', default=1)


if __name__ == '__main__':
    flag.parse_commandline(sys.argv[1:])
    flag.die_on_missing_required()

    print 'other_module.multiply_by(%d) = %d' % (
        FLAGS.some_int,
        other_module.multiply_by(FLAGS.some_int))
```

other_module.py:

```
from oscar import flag

FLAGS = flag.namespace(__name__)
FLAGS.multiplier = flag.Int('some integer', default=flag.REQUIRED)

def multiply_by(i):
    return i * FLAGS.multiplier
```

shell:

```
$ python example.py
Missing required flags:
        [other_module.]multiplier
Usage of example.py:
__main__:
        [__main__.]some_int=None: some integer value

other_module:
        [other_module.]multiplier=<required>: some integer

# Note the namespaced reference --other_module.multiplier.
```

```
$ python example.py --other_module.multiplier=2 --some_int=3
other_module.multiply_by(3) = 6
```

License

Contents:

2.1 Introduction

At Oscar, we wanted to address settings and configuration with a system that would improve operability, solve the need to change values deep in the code-base, and consume a variety of sources (i.e. command-line arguments, environment variables, configuration files). `oscar.flag` is not a module for creating user-friendly command-line interfaces, but it is a module for faceting a large code-base for configuration.

2.1.1 Goals

- Allow any module-level constants and values to be configurable. Any constant value should be a flag, and there should be no penalty for doing so.
- Eliminate collisions and reduce clutter by providing namespaces. Each flag should be namespaced to its containing module.
- Move flags in code close to the places they are used. Avoid central "config" or "settings" modules. Flags should be consumed in code similar to module constants.

Listing 2.1: `utils.aurora`

```
ZK_ENSEMBLE = 'zookeeper-1:2181,zookeeper-2:2181,zookeeper-3:2181'
# Becomes this.
FLAGS.zk_ensemble = oscar.flag.String('zookeeper ensemble')
```

- Provide clarity from the outside (i.e. command-line). Where a value/flag is used in code should be in the flag name.

```
# We know immediately that this value is set to a flag that resides in utils.aurora
$ ps u | grep 'zookeeper-1'
ian  72838   0.0  0.0  2423356    240 s003  R+    6:31PM   0:00.00 python my_app.py --utils.aurc
```

- Allow flags to be set and read from anywhere.

Listing 2.2: `test.utils.aurora`

```
    import utils.aurora

    # Run tests against local zookeeper.
    utils.aurora.FLAGS.zk_ensemble = 'localhost:2181'
```

2.1.2 Non-Goals

- Provide a set of tools for creating command-line user interfaces. Services and applications are managed through automation (e.g. configuration management).
- Validate flag values. Flag values must marshall, but no further validation is provided.

2.2 Installation and Requirements

2.2.1 Installation

Install using *pip* or *easy_install*. `oscar.flags` has zero required external dependencies.

```
$ pip install oscar.flag
```

2.2.2 Requirements

Python 2.6 and 2.7 are supported.

2.3 Usage

2.3.1 Declaring Flags

Many constant values should be declared as flags. Obtained a *`NamespaceFlagSet`* from the *`GLOBAL_FLAGS`* singleton *`GlobalFlagSet`* object via the *`namespace()`* function.

Listing 2.3: Create namespaced `FLAGS` object.

```
from oscar import flag

FLAGS = flag.namespace(__name__)
```

There should not be a need to create a *`GlobalFlagSet`* object manually in typical usage. Flag primitives predefined in the flag module:

class `oscar.flag.`**`String`**(*description*, *default=None*, *secure=False*)
    String-valued flag.

class `oscar.flag.`**`Int`**(*description*, *default=None*, *secure=False*)
    Integer-valued flag.

class `oscar.flag.`**`Float`**(*description*, *default=None*, *secure=False*)
    Float-valued flag.

class `oscar.flag.Bool` (*description*, *default=None*, *secure=False*)
Boolean-valued flag.

class `oscar.flag.List` (*inner_type*, *separator*, *description*, *default=None*, *secure=False*)
Flag that is a list of another flag type.

These are all constructed with at least a description. `default` and `secure` are optional.

Flags must be declared at module level, and can only be declared once. Redefining a flag results in an error.

Listing 2.4: Example primitive flags.

```
FLAGS.some_int = flag.Int('some int value')
FLAGS.some_string = flag.String('some string value', 'default string')
FLAGS.some_bool = flag.Bool('secure bool (why?)', secure=True)
```

Required values are indicated by setting `default` to *`REQUIRED`*.

Listing 2.5: Required flag example.

```
FLAGS.required_float = flag.Float('some required float', flag.REQUIRED)
```

Finally, *`List`* is provided which can be used to define a flag which is a list (cannot be polymorphic). It requires two additional arguments: a *`Var`* subclass defining the primitive type, and a separator:

Listing 2.6: List flag example.

```
FLAGS.int_list = flag.List(flag.Int, ',', 'a list of integer values')
```

2.3.2 Using flag values in Python

Getting

Flag values can be read directly from the *`NamespaceFlagSet`* object (`FLAGS` above).

Listing 2.7: Accessing flag values.

```
FLAGS.some_int = flag.Int('some int value')

print FLAGS.some_int * 10
```

Setting

Setting is almost the same, but values should be parseable strings, not raw values (since the setter is how the various parsers actually set the values).

```
>>> FLAGS.some_int = '42'
>>> FLAGS.some_int
42
```

Positional Arguments

Finally, the flag module may expose positional arguments if a command-line was parsed. These are available via *`args()`*:

`oscar.flag.`**`args`**`()`

Return positional `args` from *GLOBAL_FLAGS*.

Return type list[str]

2.3.3 Parsing in `__main__`

There are three parsers provided, a command-line parser, an environment parser and a parser based on `ConfigParser`.

`oscar.flag.`**`parse_commandline`**`(`*args*`)`

Parse commandline `args` with *GLOBAL_FLAGS*.

`oscar.flag.`**`parse_environment`**`(`*args*`)`

Parse environment `args` with *GLOBAL_FLAGS*.

`oscar.flag.`**`parse_ini`**`(`*file_p*`)`

Parse a `ConfigParser` compatible file with *GLOBAL_FLAGS*.

Flag parsing must be done explicitly. Each parser can be used independently or with another parser. It is suggested to use the environment parser first followed by the command-line parser, since the environment parser can read SECURED_ settings, and flags can further be overridden by the command line.

Listing 2.8: Parser Usage.

```python
import os
import sys

from oscar import flag

if __name__ == '__main__':
    flag.parse_environment(os.environ.items())
    flag.parse_commandline(sys.argv[1:])
```

The `ConfigParser` parser requires an object providing `readline()`, which includes a standard opened file.

Note that required flags must be explicitly checked via *`die_on_missing_required()`*. If a config file is read, but the path to that config file can be set on the command line, it is useful to refrain from checking if required flags have been set until after the config file is parsed.

Listing 2.9: Config-file flag.

```python
import os
import sys

from oscar import flag

FLAGS = flag.namespace(__name__)
FLAGS.config_file = flag.String('path to config file')

if __name__ == '__main__':
    flag.parse_commandline(sys.argv[1:])
    if FLAGS.config_file:
        with open(FLAGS.config_file) as config:
            flag.parse_ini(config)
    die_on_missing_required()
```

2.3.4 Setting Flags from The Outside

Short Names and Full Names

All flags are fully namespaced and are available by referencing them through their module path. For instance, if a flag `baz` is declared in module `foo.bar`, it can be referenced on the command line or in the environment through `foo.bar.baz`.

As a convenience, any uniquely named flag can be referenced on the command line or in the environment through a short name. The short name is for convenience and should not be used in scripts or configuration files. Referencing an ambiguous short name will raise a `KeyError`.

Environment Variables

If *`parse_environment()`* is called on `os.environ.items()`, environment variables will be mapped onto flags. As noted, the environment parser supports short and full names.

The environment parser will recognize SECURED_SETTING_* environment variables. These are base64 decoded and set on the appropriate flag if present (while also setting the `secure` attribute on the flag to `True`).

The environment parser will ignore extraneous environment variables that do not map to a flag.

Note: The last parse method called may overwrite flags set by previous parse methods. It is probably preferable to call *`parse_environment()`* before calling *`parse_commandline()`*.

Command Line

Command-line flags are expected to precede any positional arguments. The presence of a single "–" argument can be used to denote the end of command-line flags. A single "-" or a double "–" are equivalent in denoting a flag. A flag and its value can be separated into separate arguments (i.e. via whitespace on the command line) or a single "=".

Listing 2.10: Command-line examples.

```
# A single '-' is acceptable.
$ ./my_bin.pex -foo=bar
$ ./my_bin.pex -foo bar
# A double '--' is also acceptable.
$ ./my_bin.pex --foo=bar
$ ./my_bin.pex --foo bar
# Everything after '--' becomes a positional argument.
$ ./my_bin.pex -- --foo --bar --baz
```

`Bool` flags are special-cased in the command-line parser. A standalone *`Bool`* flag with no value indicates `True`. A *`Bool`* flag can only be set explicitly using "=", and a space between the flag and the value is invalid. Valid boolean values are any case folded variation of `yes`, `true`, `on`, `1` for `True`, and `no`, `false`, `off`, `0` for `False`. Other values result in an error.

Listing 2.11: Boolean special case command-line parsing.

```
./my_bin.pex --some_bool  # some_bool is set to True
./my_bin.pex --some_bool=False  # some_bool is set to False
./my_bin.pex --some_bool False  # this is invalid and will throw a parse error
```

Command-line flags also may be referenced by their short name if it's non-ambiguous, though this is provided as a shortcut for users, and the fully qualified flag name should be used in configuration and scripting.

Finally, if any flags are encountered that do not map to a flag in the application, an error will be raised. All command-line flags must map to a declared flag.

`ConfigParser` ini files

There is also support for ini files. Sections map to namespaces, and key/value pairs within the sections map to flags within those namespaces.

Listing 2.12: example.ini

```
[__main__]
output=output.txt

[utils.zookeeper]
ensemble=zookeeper-1,zookeeper-2,zookeeper-3
```

The ini parser will need to be ran against a file-like object.

Listing 2.13: Using *parse_ini()*.

```
with open('~/.config.ini') as config:
    flag.parse_ini(config)
```

Finally, every section and key/value within an ini file must map to a namespace and flag. Unexpected sections and keys will raise an error.

2.3.5 Additional Public API for flag

`oscar.flag.`**GLOBAL_FLAGS = <oscar.flag.GlobalFlagSet object>**

GlobalFlagSet is a collection of namespaces and flag logic.

`GLOBAL_FLAGS.`**usage**

This attribute can be replaced with a function that will print usage (invoked automatically by –help on the command line). The function accepts a single parameter: the *GlobalFlagSet* object calling it. The default implementation calls *GlobalFlagSet.write_flags()*. Default value is *default_usage()*.

`GLOBAL_FLAGS.`**usage_long**

This attribute can be replaced with a function that will print long usage (invoked automatically by –helplong on the command line). The function accepts a single parameter: the *GlobalFlagSet* object calling it. The default implementation calls *GlobalFlagSet.write_flags_long()*. Default value is *default_usage_long()*.

`GlobalFlagSet.`**visit**(*func*)

Walk all *set* flags, calling `func` on each.

Parameters **func** (*F(str, str, Var)*) – visiting function

GlobalFlagSet.**visit_all**(*func*)

Walk all flags, calling `func` on each.

Parameters **func** (*F(str, str, Var)*) – visiting function

2.4 oscar.flag

oscar.flag.**GLOBAL_FLAGS = <oscar.flag.GlobalFlagSet object>**

GlobalFlagSet is a collection of namespaces and flag logic.

oscar.flag.**REQUIRED = <oscar.flag._Required object>**

Setting a flag's `default` to this constant marks it as required.

class oscar.flag.**Bool**(*description*, *default=None*, *secure=False*)

Boolean-valued flag.

set(*value*)

type_str = 'Bool'

exception oscar.flag.**FlagException**

Error in flag initialization or access.

class oscar.flag.**Float**(*description*, *default=None*, *secure=False*)

Float-valued flag.

set(*value*)

type_str = 'Float'

class oscar.flag.**GlobalFlagSet**(*usage=<function default_usage>*, *usage_long=<function default_usage_long>*)

GlobalFlagSet is a collection of namespaces and flag logic.

check_required()

Returns a list of `(namespace, name, flag)` of all unset, required flags.

Return type list[tuple(str, str, Var)]

find_short(*flag*)

Find the namespace for a non-qualified `flag`.

If the `flag` is not found or multiple flags are found, `KeyError` is raised.

Return type str

Raises **KeyError** – if the flag is not found or ambiguous

get(*namespace*, *flag*)

get the flag object associated with `flag` in `namespace`.

Return type *Var*

Raises **KeyError** – if the flag is not found

namespace(*namespace*)

Returns a *NamespaceFlagSet* associated with `namespace`.

Return type *NamespaceFlagSet*

parse_commandline(*args*)

Parse a commandline into flags and arguments.

Parse a commandline. a single '-' and a double '–' are treated as equivalent for denoting a flag. Flag values may be separated by '=' or be the next argument. Booleans are a special case that indicate a true value or must have their values separated by '='.

Raises

- **`KeyError`** – on unknown flag
- **`ParseException`** – on invalid command-line syntax

`parse_environment`(*args*)

Parse environment variable tuples.

Call with os.environ:

```
>>> import os
>>> flag.parse_environment(os.environ.items())
```

Recognizes *SECURED_SETTING_* prefixed flags, translates from base64 and maps to the short name. *secure* is also set to *True* on the underlying flag object, which should be respected by users of *`visit()`* and *`visit_all()`*.

`parse_ini`(*file_p*)

Parse a `ConfigParser` compatible file object.

Namespaces are section headers, keys are flags:

```
[__main__]
foo=bar

[foo.bar]
baz=42
```

`file_p` only needs to implement `readline(size=0)`.

`visit`(*func*)

Walk all *set* flags, calling `func` on each.

Parameters **`func`** (*`F(str, str, Var)`*) – visiting function

`visit_all`(*func*)

Walk all flags, calling `func` on each.

Parameters **`func`** (*`F(str, str, Var)`*) – visiting function

`write_flags`(*out*, *namespace='__main__'*)

Prints the usage to `out`.

`write_flags_long`(*out*)

Prints all flag usage to `out`.

class `oscar.flag.`**`Int`**(*description*, *default=None*, *secure=False*)

Integer-valued flag.

`set`(*value*)

`type_str` = **'Int'**

class `oscar.flag.`**`List`**`(`*inner_type*, *separator*, *description*, *default=None*, *secure=False*`)`
Flag that is a list of another flag type.

`set``(`*value*`)`

`type_str` = **'List[_]'**

class `oscar.flag.`**`NamespaceFlagSet`**
Represents a set of flags in a namespace.

exception `oscar.flag.`**`ParseException`**
Error in command-line parsing.

class `oscar.flag.`**`String`**`(`*description*, *default=None*, *secure=False*`)`
String-valued flag.

`set``(`*value*`)`

`type_str` = **'String'**

class `oscar.flag.`**`Var`**`(`*description*, *default=None*, *secure=False*`)`
Base of all flag accessors.

`get``()`
Return the flag value, or default if it is not set.

`is_set``()`

Return type bool

`type_str` = **'Unknown'**

`value` = **<object object>**

`oscar.flag.`**`args`**`()`
Return positional `args` from *`GLOBAL_FLAGS`*.

Return type list[str]

`oscar.flag.`**`default_usage`**`(`*globalflags*`)`
Default for printing out usage.

`oscar.flag.`**`default_usage_long`**`(`*globalflags*, *return_code=0*`)`
Default for printing out long-form usage.

`oscar.flag.`**`die_on_missing_required`**`()`
If missing required flags, die and write usage.

`oscar.flag.`**`namespace`**`(`*name*`)`
Return a namespace from *`GLOBAL_FLAGS`*.

Return type *NamespaceFlagSet*

`oscar.flag.`**`parse_commandline`**`(`*args*`)`
Parse commandline `args` with *`GLOBAL_FLAGS`*.

`oscar.flag.`**`parse_environment`**`(`*args*`)`
Parse environment `args` with *`GLOBAL_FLAGS`*.

`oscar.flag.`**`parse_ini`**(*file_p*)

Parse a `ConfigParser` compatible file with *`GLOBAL_FLAGS`*.

2.5 oscar.flag.contrib

The `oscar.flag.contrib` contains additional non-core functionality. This module may introduce optional dependencies for usage.

class `oscar.flag.contrib.`**`Datetime`**(*description*, *default=None*, *secure=False*)

Datetime-valued flag.

CHAPTER 3

Indices and tables

- genindex
- search
- modindex

Python Module Index

Index