blargs Documentation

Release 0.2.22a

Karl Gyllstrom

May 22, 2012

CONTENTS

blargs provides easy command line parsing, as an alternative to argparse and optparse from Python's standard library. The main distinctions are:

- Cleaner, more minimal, and possibly more *pythonic* syntax.
- Support for arbitrarily complex dependency relationships. For example, argument A might require arguments B and C, while conflicting with D; or requiring argument E in the case that A is less than 10 or B is equal to the string 'wonderful!'.
- Emphasis on *ease of use* over *configurability*.

Blargs has been tested on Python2.6 to Python3.2 and PyPy.

Note: blargs is currently still in *beta*. You can help by submitting bugs here!

CHAPTER

ONE

INSTALLATION

By Pip:

```
pip install blargs
```

Or by git:

```
git clone https://bitbucket.org/gyllstromk/blargs.git
```

CHAPTER

TWO

LICENSE

BSD

CHAPTER

THREE

QUICK START

The preferred use of `Parser` is via the `with` idiom, as follows:

```
>>> with Parser(locals()) as p:
...     p.int('arg1')
...     p.str('arg2')
...     p.flag('arg3')
...
>>> print 'Out of with statement; sys.argv is now parsed!'
>>> print arg1, arg2, arg3
```

Note the use of `locals` is limited to the global scope; use a dictionary otherwise, getting argument values using the argument names as keys.

The user can now specify the following command lines:

```
python test.py --arg1=3          # either '=' ...
python test.py --arg1 3          # ...or space is allowed
python test.py --arg2 'hello'    #
python test.py --arg3            # no value is specified; implied true
```

The following command lines will be rejected:

```
python test.py --arg1    # no value specified for 'arg1'
python test.py --arg1 a  # 'a' does not parse to int
python test.py --arg3 a  # 'arg3' is a flag and does not accept a value
```

Additionally, users can query for help:

```
python test.py --help
```

To which the program will respond:

```
Arguments:
    --arg1 <int>
    --arg2 <option>
    --arg3
    --help/-h
```

CHAPTER

FOUR

SPECIFYING ARGUMENTS

```
>>> with Parser(locals()) as p:
...     # basic types
...     p.str('string_arg')       # --string_arg='hello'
...     p.int('int_arg')          # --int_arg 3
...     p.float('float_arg')      # --float_arg 9.6
...     p.flag('flag_arg')        # --flag_arg  (no argument passed)
...     # complex types
...     p.range('range_arg')      # --range_arg 1:2
...     p.multiword('multi_arg') # --multi_arg hello world
...     p.file('file_arg')        # --file_arg README.txt
...     p.directory('dir_arg')    # --dir_arg /tmp/
```

On occasions you may need to refer to a created argument to specify relationships. This can be done at creation time, or by a lookup. The following:

```
>>> with Parser(locals()) as p:
...     argument1 = p.str('arg1')
```

is equivalent to:

```
>>> with Parser(locals()) as p:
...     p.str('arg1')
...     argument1 = p['arg1']
```

Note that `argument1` *does not* get the value of the parsed value; it represents the argument object itself.

CHAPTER

FIVE

HOWTO

5.1 Require an argument

```
>>> with Parser(locals()) as p:
...     p.str('required_arg').required()
```

If we try to not pass it:

```
python test.py
```

We get the following:

```
No value passed for required_arg
usage: test.py [--required_arg <option>] [--help,-h]
```

5.2 Shorthand/alias

```
>>> with Parser(locals()) as p:
...     p.str('arg1').shorthand('a')
```

Either is acceptable:

```
python test.py --arg1 my_string
python test.py -a my_string
```

Note that we can specify any number of attributes by daisy-chaining calls. For example:

```
>>> with Parser(locals()) as p:                 # 'arg1' has 'a' as alias and
...     p.str('arg1').shorthand('a').required()  # is also required
```

5.3 Dependencies/Conflicts

Dependencies indicate that some argument is required if another one is specified, while conflicts indicate that two or more arguments may not be mutually specified.

```
>>> with Parser(locals()) as p:                 # if 'arg2' is specified,
...     arg1 = p.str('arg1')                    # so too must be 'arg3'
...     p.str('arg2').requires(                 # and 'arg1'. Note: if 'arg1'
...       p.str('arg3'),                        # is specified, this does not
```

```
...       arg1,                                    # mean 'arg2' must be
...     )
...     p.str('arg4').conflicts(                   # if 'arg4' is specified, then
...       arg1                                     # 'arg1' may not be.
...     )
```

A slightly more complex example:

```
>>> with Parser(locals()) as p:
...     p.float('arg1').requires(  # if 'arg1' is specified
...       p.int('arg2'),           # then both 'arg2'
...       p.flag('arg3'),          # and 'arg3' must be too, however,
...     ).conflicts(               # if it is specified,
...       p.str('arg4'),           # then neither 'arg4'
...       p.range('arg5')          # nor 'arg5' may be specified
...     )
```

5.4 Allowing Duplicates/Multiple

Normally an argument may only be specified once by the user. This can be changed:

```
>>> with Parser(locals()) as p:
...     p.str('arg1').multiple()
...
>>> print len(arg1)  # arg1 is list
>>> print arg1[0]
```

To use:

```
python test.py --arg1 hello --arg1 world
```

Now the value of arg1 is `['hello', 'world']`.

Note: by indicating `multiple`, the variable is stored as a `list` *even* if only one instance is specified by the user.

5.5 Indicating default values or environment variables

A default value means the argument will receive the value if not specified.

```
>>> with Parser(locals()) as p:
...     p.str('arg1').default('hello')
```

Both executions are equivalent:

```
python test.py --arg1 hello
python test.py
```

Additionally, we can specify that an argument should be drawn from the OS/shell environment if not provided at the command line:

```
>>> with Parser(locals()) as p:
...     p.str('port').environment()
```

Now the following shell interactions are equivalent:

```
python test.py --port 5000
export PORT=5000; python test.py
```

Currently, this works by setting the default value to the environment value. Consequently, the *default* and *environment* arguments currently conflict (or, specifically, *override* one another).

5.6 Allowing no argument label

If we want an argument to be parsed even without a label:

```
>>> with Parser(locals()) as p:
...     p.str('arg1').unspecified_default()
...     p.str('arg2')
```

Now, an argument without a label will be saved to `arg1`:

```
python test.py hello  # arg1 = 'hello'
python test.py --arg2 world hello   # arg1 = 'hello', arg2 = 'world'
```

Note that to avoid ambiguity, only one argument type may be an `unspecified_default`.

5.7 Creating your own types

It is possible to create your own types using the *cast* function, in which you specify a function that is run on the value at parse time. Let's say we want the user to be able to pass a comma-separated list of `float` values, or a space-delimited list of `int` values:

```
>>> with Parser(locals()) as p:
...     p.str('floatlist').cast(lambda x: [float(val) for val in x.split(',')])
...     p.multiword('intlist').cast(lambda x: [int(val) for val in x.split()])
```

A sample command line:

```
python test.py --floatlist 1.2,3.9,8.6 --intlist 1 9 2
```

We now can access these:

```
>>> print floatlist, intlist
... [1.2, 3.9, 8.6], [1, 9, 2]
```

CHAPTER

SIX

CONDITIONS

Conditions extend the concept of dependencies and conflicts with conditionals.

6.1 if

Argument must be specified if `condition`:

```
>>> with Parser(locals()) as p:
...     arg1 = p.int('arg1')
...     arg2 = p.float('arg2').if_(arg1 > 10) # 'arg2' must be specified if
...                                           # 'arg1' > 10
...     p.float('arg3').if_(arg1.or_(arg2))   # 'arg3' must be specified if
...                                           # 'arg1' or 'arg2' is
```

6.2 unless

Argument must be specified `unless condition`.

```
>>> with Parser(locals()) as p:
...     arg1 = p.int('arg1')
...     p.float('arg2').unless(arg1 > 10)     # 'arg2' must be specified if
...                                           # 'arg1' <= 10
```

6.3 requires

We described `requires` previously, but here we show that it also works with conditional expressions.

If argument is specified, then `condition` must be true;

```
>>> with Parser(locals()) as p:
...     arg1 = p.int('arg1')
...     p.float('arg2').requires(arg1 < 20)   # if 'arg2' specified, 'arg1' must
...                                           # be < 20
```

6.4 and/or

Build conditions via logical operators `and_` and `or_`:

```
>>> with Parser(locals()) as p:
...     arg1 = p.int('arg1')
...     p.float('arg2').unless((0 < arg1).and_(arg1 < 10))   # 'arg2' is required
...                                                          # unless 0 < arg1 < 10
...
...     p.float('arg3').if_((arg1 < 0).or_(arg1 > 10))       # 'arg3' is required
...                                                          # if 'arg1' < 0 or
...                                                          # 'arg1' > 10
```

CHAPTER

SEVEN

AGGREGATE CALLS

Aggregate calls enable the indication of behavior for a set of arguments at once.

7.1 At least one

Require at least one (up to all) of the subsequent arguments:

```
>>> with Parser(locals()) as p:
...     p.at_least_one(
...         p.str('arg1'),
...         p.str('arg2'),
...         p.str('arg3')
...     )
```

7.2 Mutual exclusion

Only one of the arguments can be specified:

```
>>> with Parser(locals()) as p:
...     p.only_one_if_any(
...         p.str('arg1'),
...         p.str('arg2'),
...         p.str('arg3')
...     )
```

7.3 All if any

If any of the arguments is be specified, all of them must be:

```
>>> with Parser(locals()) as p:
...     p.all_if_any(
...         p.str('arg1'),
...         p.str('arg2'),
...         p.str('arg3')
...     )
```

7.4 Require one

One and only one of the arguments must be specified:

```
>>> with Parser(locals()) as p:
...     p.require_one(
...         p.str('arg1'),
...         p.str('arg2'),
...         p.str('arg3')
...     )
```

CHAPTER

EIGHT

COMPLEX DEPENDENCIES

```
>>> with Parser(locals()) as p:
...     p.at_least_one(             # at least one of
...       p.only_one_if_any(        # 'arg1', 'arg2', and/or 'arg3'
...         p.int('arg1'),          # must be specified, but
...         p.flag('arg2'),         # 'arg1' and 'arg2' may not
...       ),                        # both be specified
...       p.str('arg3'),
...     )

>>> with Parser(locals() as p:
...      p.require_one(
...          p.all_if_any(
...              p.only_one_if_any(
...                  p.flag('a'),
...                  p.flag('b'),
...              ),
...              p.flag('c'),
...          ),
...          p.only_one_if_any(
...              p.all_if_any(
...                  p.flag('d'),
...                  p.flag('e'),
...              ),
...              p.flag('f'),
...          ),
...      )
```

Accepts these combinations:

```
a, c; b, c; d, e; f
```

CHAPTER

NINE

CUSTOMIZING

9.1 Indicating label style

By default, `--` denotes a full argument while `-` denotes the shorthand/alias variant. This can be replaced via `set_single_prefix()` and `set_double_prefix()`.

9.2 Setting help function

The `set_help_prefix()` allows you to specify the content that appears before the argument list when users trigger the `--help` command.

CHAPTER

TEN

CODE

class blargs.**Parser**(*store=None*, *default_help=True*)

Command line parser.

all_if_any(**args*)

If *any* of args is specified, then all of args must be specified.

at_least_one(**args*)

Require at least one of args.

bool(*name*)

Alias of flag().

config(*name*)

Add configuration file, whose key/value pairs will provide/replace any arguments created for this parser. For example:

```
>>> with Parser() as p:
...   p.int('a')
...   p.str('b')
...   p.config('conf')
```

Now, arg a can be specfied on the command line, or in the configuration file passed to conf. For example:

```
python test.py --a 3
python test.py --conf myconfig.cfg
```

Where myconfig.cfg:

```
[myconfigfile]
a = 5
b = 9
x = 'hello'
```

Note that any parameters in the config that aren't created as arguments via this parser are ignored. In the example above, the values of variables a and b would be assigned, while x would be ignored (as the developer did not create an x argument).

If anything is specified on the command line, its value is not taken from the config file. For example:

```
python test.py --a 3 --config myconfig.cfg
```

In this case, the value of a is 3 (from the command line) and not 5 (from the config file).

directory(*name*, *create=False*)

File directory value. Checks to ensure that the user passed file name exists and is a directory (i.e., not some

other file object). If `create` is specified, creates the directory using `os.makedirs`; any intermediate directories are also created.

file (*name*, *mode=None*, *buffering=None*)
Opens the file indicated by the name passed by the user. `mode` and `buffering` are arguments passed to `open`.

The example below implements a file copy operation:

```
>>> with Parser(locals()) as p:
...     p.file('input_file')
...     p.file('output_file', mode='w')
...
... output_file.write(input_file.read())
```

flag (*name*)
Boolean value. The presence of this flag indicates a true value, while an absence indicates false. No arguments.

float (*name*)
Add float argument.

int (*name*)
Add integer argument.

multiword (*name*)
Accepts multiple terms as an argument. For example:

```
>>> with Parser() as p:
...   p.multiword('multi')
```

Now accepts:

```
python test.py --multi path to something
```

only_one_if_any (**args*)
If *any* of `args` is specified, then none of the remaining `args` may be specified.

range (*name*)
Range type. Accepts similar values to that of python's py:*range* and py:*xrange*. Accepted delimiters are space, -, and :.

```
>>> with Parser() as p:
...   p.range('values')
```

Now accepts:

```
python test.py --values 10  # -> xrange(10)
python test.py --values 0-1  # -> xrange(0, 1)
python test.py --values 0:10:2  # -> xrange(0, 10, 2)
python test.py --values 0 10 3  # -> xrange(0, 10, 3)
```

require_one (**args*)
Require only and only one of `args`.

set_double_prefix (*flag*)
Set the double flag prefix. This appears before long arguments (e.g., –arg).

set_help_prefix (*message*)
Indicate text to appear before argument list when the `help` function is triggered.

set_single_prefix(*flag*)
Set the single flag prefix. This appears before short arguments (e.g., -a).

str(*name*)
Add `str` argument.

underscore()
Convert '-' to '_' in argument names. This is enabled if `with_locals` is used, as variable naming rules are applied.

url(*name*)
URL value; verifies that argument has a scheme (e.g., http, ftp, file).

classmethod with_locals()
Create `Parser` using locals() dict.

class `blargs.`**Option**(*argname*, *parser*)

cast(*cast*)
Provide a casting value for this argument.

conflicts(**conditions*)
Specifiy other conditions which this argument conflicts with.

Parameters conditions (sequence of either `Option` or `Condition`) – conflicting options/conditions

default(*value*)
Provide a default value for this argument.

environment()
Pull argument value from OS environment if unspecified. The case of the argument name, all lower, and all upper are all tried. For example, if the argument name is *Port*, the following names will be used for environment lookups: *Port*, *port*, *PORT*.

```
>>> with Parser(locals()) as p:
...     p.int('port').environment()
```

Both command lines work:

```
python test.py --port 5000
export PORT=5000; python test.py
```

if_(*condition*)
Argument is required if `conditions`.

multiple()
Indicate that the argument can be specified multiple times.

required()
Indicate that this argument is required.

requires(**conditions*)
Specifiy other options/conditions which this argument requires.

Parameters conditions – required conditions

shorthand(*alias*)
Set shorthand for this argument. Shorthand arguments are 1 character in length and are prefixed by a single '-'. For example:

```
>>> parser.str('option').shorthand('o')
```

would cause '–option' and '-o' to be alias argument labels when invoked on the command line.

Parameters **alias** – alias of argument

unless(*condition*)
Argument is required unless `conditions`.

unspecified_default()
Indicate that values passed without argument labels will be attributed to this argument.

10.1 Exceptions

class blargs.**ArgumentError**
Root class of all arguments that are thrown due to user input which violates a rule set by the parser. In other words, errors of this type should be caught and communicated to the user some how. The default behavior is to signal the particular error and show the *usage*.

class blargs.**FormatError**
Argument not formatted correctly.

class blargs.**MissingRequiredArgumentError**(*arg*)
Required argument not specified.

class blargs.**ManyAllowedNoneSpecifiedArgumentError**(*allowed*)
An argument out of a list of required is missing. e.g., one of -a and -b is required and neither is specified.

class blargs.**MultipleSpecifiedArgumentError**
Multiple of the same argument specified.

class blargs.**DependencyError**(*arg1*, *arg2*)
User specified an argument that requires another, unspecified argument.

class blargs.**ConflictError**(*offender1*, *offender2*)
User specified an argument that conflicts another specified argument.

class blargs.**UnspecifiedArgumentError**(*arg*)
User supplies argument that isn't specified.