argparse — Command-Line Option and Argument Parsing

Purpose: Command-line option and argument parsing.

The argparse module includes tools for building command line argument and option processors. It was added to Python 2.7 as a replacement for optparse. The implementation of argparse supports features that would not have been easy to add to optparse, and that would have required backwards-incompatible API changes, so a new module was brought into the library instead. optparse is now deprecated.

Setting Up a Parser

The first step when using argparse is to create a parser object and tell it what arguments to expect. The parser can then be used to process the command-line arguments when the program runs. The constructor for the parser class (ArgumentParser) takes several arguments to set up the description used in the help text for the program and other global behaviors or settings.

```
import argparse
parser = argparse.ArgumentParser(
    description='This is a PyMOTW sample program',
)
```

Defining Arguments

argparse is a complete argument processing library. Arguments can trigger different actions, specified by the action argument to add_argument(). Supported actions include storing the argument (singly, or as part of a list), storing a constant value when the argument is encountered (including special handling for true/false values for Boolean switches), counting the number of times an argument is seen, and calling a callback to use custom processing instructions.

The default action is to store the argument value. If a type is provided, the value is converted to that type before it is stored. If the dest argument is provided, the value is saved using that name when the command-line arguments are parsed.

Parsing a Command-Line

After all of the arguments are defined, parse the command-line by passing a sequence of argument strings to parse_args(). By default, the arguments are taken from sys.argv[1:], but any list of strings can be used. The options are processed using the GNU/POSIX syntax, so option and argument values can be mixed in the sequence.

The return value from parse_args() is a Namespace containing the arguments to the command. The object holds the argument values as attributes, so if the argument's dest is set to "myoption", the value is accessible as args.myoption.

Simple Examples

Here is a simple example with three different options: a Boolean option (-a), a simple string option (-b), and an integer option (-c).

```
# argparse_short.py
```

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args(['-a', '-bval', '-c', '3']))
```

There are a few ways to pass values to single character options. The previous example uses two different forms, -bval and -c val.

```
$ python3 argparse_short.py

Namespace(a=True, b='val', c=3)
```

The type of the value associated with `'c'` in the output is an integer, since the `ArgumentParser` was told to convert the argument before storing it.

"Long" option names, with more than a single character in their name, are handled in the same way.

```
# argparse_long.py
```

```
import argparse

parser = argparse.ArgumentParser(
    description='Example with long option names',
)

parser.add_argument('--noarg', action="store_true",
                    default=False)
parser.add_argument('--witharg', action="store",
                    dest="witharg")
parser.add_argument('--witharg2', action="store",
                    dest="witharg2", type=int)

print(
    parser.parse_args(
        ['--noarg', '--witharg', 'val', '--witharg2=3']
    )
)
```

The results are similar.

```
$ python3 argparse_long.py

Namespace(noarg=True, witharg='val', witharg2=3)
```

`argparse` is a full command-line argument parser tool, and handles both optional and required arguments.

```
# argparse_arguments.py
```

```
import argparse

parser = argparse.ArgumentParser(
    description='Example with nonoptional arguments',
)

parser.add_argument('count', action="store", type=int)
parser.add_argument('units', action="store")

print(parser.parse_args())
```

In this example, the "count" argument is an integer and the "units" argument is saved as a string. If either is left off of the command-line, or the value given cannot be converted to the right type, an error is reported.

```
$ python3 argparse_arguments.py 3 inches

Namespace(count=3, units='inches')

$ python3 argparse_arguments.py some inches

usage: argparse_arguments.py [-h] count units
argparse_arguments.py: error: argument count: invalid int value:
'some'

$ python3 argparse_arguments.py

usage: argparse_arguments.py [-h] count units
argparse_arguments.py: error: the following arguments are
required: count, units
```

Argument Actions

Any of six built-in actions can be triggered when an argument is encountered.

`store`
: Save the value, after optionally converting it to a different type. This is the default action taken if none is specified explicitly.

`store_const`
: Save a value defined as part of the argument specification, rather than a value that comes from the arguments being parsed. This is typically used to implement command-line flags that are not Booleans.

`store_true` / `store_false`
: Save the appropriate Boolean value. These actions are used to implement Boolean switches.

`append`
: Save the value to a list. Multiple values are saved if the argument is repeated.

`append_const`
: Save a value defined in the argument specification to a list.

`version`
: Prints version details about the program and then exits.

This example program demonstrates each action type, with the minimum configuration needed for each to work.

```python
# argparse_action.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-s', action='store',
                    dest='simple_value',
                    help='Store a simple value')

parser.add_argument('-c', action='store_const',
                    dest='constant_value',
                    const='value-to-store',
                    help='Store a constant value')

parser.add_argument('-t', action='store_true',
                    default=False,
                    dest='boolean_t',
                    help='Set a switch to true')
parser.add_argument('-f', action='store_false',
                    default=True,
                    dest='boolean_f',
                    help='Set a switch to false')

parser.add_argument('-a', action='append',
                    dest='collection',
                    default=[],
                    help='Add repeated values to a list')

parser.add_argument('-A', action='append_const',
                    dest='const_collection',
                    const='value-1-to-append',
                    default=[],
                    help='Add different values to list')
parser.add_argument('-B', action='append_const',
                    dest='const_collection',
                    const='value-2-to-append',
                    help='Add different values to list')

parser.add_argument('--version', action='version',
                    version='%(prog)s 1.0')

results = parser.parse_args()
print('simple_value     = {!r}'.format(results.simple_value))
print('constant_value   = {!r}'.format(results.constant_value))
print('boolean_t        = {!r}'.format(results.boolean_t))
print('boolean_f        = {!r}'.format(results.boolean_f))
print('collection       = {!r}'.format(results.collection))
print('const_collection = {!r}'.format(results.const_collection))
```

The -t and -f options are configured to modify different option values, each storing either True or False. The dest values for -A and -B are the same so that their constant values are appended to the same list.

```
$ python3 argparse_action.py -h

usage: argparse_action.py [-h] [-s SIMPLE_VALUE] [-c] [-t] [-f]
                          [-a COLLECTION] [-A] [-B] [--version]

optional arguments:
  -h, --help        show this help message and exit
  -s SIMPLE_VALUE   Store a simple value
  -c                Store a constant value
  -t                Set a switch to true
  -f                Set a switch to false
  -a COLLECTION     Add repeated values to a list
  -A                Add different values to list
  -B                Add different values to list
  --version         show program's version number and exit

$ python3 argparse_action.py -s value

simple_value     = 'value'
constant_value   = None
boolean_t        = False
boolean_f        = True
collection       = []
const_collection = []

$ python3 argparse_action.py -c

simple_value     = None
constant_value   = 'value-to-store'
boolean_t        = False
boolean_f        = True
collection       = []
const_collection = []

$ python3 argparse_action.py -t

simple_value     = None
constant_value   = None
boolean_t        = True
boolean_f        = True
collection       = []
const_collection = []

$ python3 argparse_action.py -f

simple_value     = None
constant_value   = None
boolean_t        = False
boolean_f        = False
collection       = []
const_collection = []

$ python3 argparse_action.py -a one -a two -a three

simple_value     = None
constant_value   = None
boolean_t        = False
boolean_f        = True
collection       = ['one', 'two', 'three']
const_collection = []

$ python3 argparse_action.py -B -A

simple_value     = None
constant_value   = None
boolean_t        = False
boolean_f        = True
```

```
collection       = []
const_collection = ['value-2-to-append', 'value-1-to-append']

$ python3 argparse_action.py --version

argparse_action.py 1.0
```

Option Prefixes

The default syntax for options is based on the Unix convention of signifying command-line switches using a dash prefix (“-“). argparse supports other prefixes, so a program can conform to the local platform default (i.e., use “/” on Windows) or follow a different convention.

```
# argparse_prefix_chars.py

import argparse

parser = argparse.ArgumentParser(
    description='Change the option prefix characters',
    prefix_chars='-+/',
)

parser.add_argument('-a', action="store_false",
                    default=None,
                    help='Turn A off',
                    )
parser.add_argument('+a', action="store_true",
                    default=None,
                    help='Turn A on',
                    )
parser.add_argument('//noarg', '++noarg',
                    action="store_true",
                    default=False)

print(parser.parse_args())
```

Set the `prefix_chars` parameter for the `ArgumentParser` to a string containing all of the characters that should be allowed to signify options. It is important to understand that although `prefix_chars` establishes the allowed switch characters, the individual argument definitions specify the syntax for a given switch. This gives explicit control over whether options using different prefixes are aliases (such as might be the case for platform-independent command-line syntax) or alternatives (e.g., using “+” to indicate turning a switch on and “-” to turn it off). In the previous example, +a and -a are separate arguments, and //noarg can also be given as ++noarg, but not --noarg.

```
$ python3 argparse_prefix_chars.py -h

usage: argparse_prefix_chars.py [-h] [-a] [+a] [//noarg]

Change the option prefix characters

optional arguments:
  -h, --help        show this help message and exit
  -a                Turn A off
  +a                Turn A on
  //noarg, ++noarg

$ python3 argparse_prefix_chars.py +a

Namespace(a=True, noarg=False)

$ python3 argparse_prefix_chars.py -a

Namespace(a=False, noarg=False)

$ python3 argparse_prefix_chars.py //noarg

Namespace(a=None, noarg=True)

$ python3 argparse_prefix_chars.py ++noarg

Namespace(a=None, noarg=True)
```

```
$ python3 argparse_prefix_chars.py --noarg

usage: argparse_prefix_chars.py [-h] [-a] [+a] [//noarg]
argparse_prefix_chars.py: error: unrecognized arguments: --noarg
```

Sources of Arguments

In the examples so far, the list of arguments given to the parser has come from a list passed in explicitly, or were taken implicitly from `sys.argv`. Passing the list explicitly is useful when using `argparse` to process command-line-like instructions that do not come from the command-line (such as in a configuration file).

```
# argparse_with_shlex.py
```

```
import argparse
from configparser import ConfigParser
import shlex

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

config = ConfigParser()
config.read('argparse_with_shlex.ini')
config_value = config.get('cli', 'options')
print('Config  :', config_value)

argument_list = shlex.split(config_value)
print('Arg List:', argument_list)

print('Results :', parser.parse_args(argument_list))
```

This example uses `configparser` to read a configuration file.

```
[cli]
options = -a -b 2
```

`shlex` makes it easy to split the string stored in the configuration file.

```
$ python3 argparse_with_shlex.py

Config  : -a -b 2
Arg List: ['-a', '-b', '2']
Results : Namespace(a=True, b='2', c=None)
```

An alternative to processing the configuration file in application code is to tell `argparse` how to recognize an argument that specifies an input file containing a set of arguments to be processed using `fromfile_prefix_chars`.

```
# argparse_fromfile_prefix_chars.py
```

```
import argparse
import shlex

parser = argparse.ArgumentParser(description='Short sample app',
                                 fromfile_prefix_chars='@',
                                 )

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args(['@argparse_fromfile_prefix_chars.txt']))
```

This example stops when it finds an argument prefixed with `@`, then reads the named file to find more arguments. The file should contain one argument per line, as in this example.

```
# argparse_fromfile_prefix_chars.txt
```

```
-a
-b
2
```

The output produced when processing `argparse_from_prefix_chars.txt` follows.

```
$ python3 argparse_fromfile_prefix_chars.py

Namespace(a=True, b='2', c=None)
```

Help Output

Automatically Generated Help

`argparse` will automatically add options to generate help, if configured to do so. The `add_help` argument to `ArgumentParser` controls the help-related options.

```
# argparse_with_help.py

import argparse

parser = argparse.ArgumentParser(add_help=True)

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args())
```

The help options (`-h` and `--help`) are added by default, but can be disabled by setting `add_help` to false.

```
# argparse_without_help.py

import argparse

parser = argparse.ArgumentParser(add_help=False)

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args())
```

Although `-h` and `--help` are defacto standard option names for requesting help, some applications or uses of `argparse` either do not need to provide help or need to use those option names for other purposes.

```
$ python3 argparse_with_help.py -h

usage: argparse_with_help.py [-h] [-a] [-b B] [-c C]

optional arguments:
  -h, --help  show this help message and exit
  -a
  -b B
  -c C

$ python3 argparse_without_help.py -h

usage: argparse_without_help.py [-a] [-b B] [-c C]
argparse_without_help.py: error: unrecognized arguments: -h
```

Customizing Help

For applications that need to handle the help output directly, some of the utility methods of `ArgumentParser` will be useful in creating custom actions to print help with extra information.

```
# argparse_custom_help.py

import argparse

parser = argparse.ArgumentParser(add_help=True)

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print('print_usage output:')
parser.print_usage()
print()

print('print_help output:')
parser.print_help()
```

`print_usage()` prints the short usage message for an argument parser, and `print_help()` prints the full help output.

```
$ python3 argparse_custom_help.py

print_usage output:
usage: argparse_custom_help.py [-h] [-a] [-b B] [-c C]

print_help output:
usage: argparse_custom_help.py [-h] [-a] [-b B] [-c C]

optional arguments:
  -h, --help  show this help message and exit
  -a
  -b B
  -c C
```

The `ArgumentParser` uses a formatter class to control the appearance of the help output. To change the class, pass `formatter_class` when instantiating the `ArgumentParser`.

For example, the `RawDescriptionHelpFormatter` bypasses the line wrapping provided by the default formatter.

```
# argparse_raw_description_help_formatter.py

import argparse

parser = argparse.ArgumentParser(
    add_help=True,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    description="""
    description
        not
           wrapped""",
    epilog="""
    epilog
      not
         wrapped""",
)

parser.add_argument(
    '-a', action="store_true",
    help="""argument
    help is
    wrapped
    """,
)

parser.print_help()
```

All text in the description and epilog of the command will be left unchanged.

```
$ python3 argparse_raw_description_help_formatter.py
```

```
usage: argparse_raw_description_help_formatter.py [-h] [-a]

    description
        not
           wrapped

optional arguments:
  -h, --help  show this help message and exit
  -a          argument help is wrapped

    epilog
      not
         wrapped
```

The `RawTextHelpFormatter` treats all help text as pre-formatted.

```
# argparse_raw_text_help_formatter.py
```

```
import argparse

parser = argparse.ArgumentParser(
    add_help=True,
    formatter_class=argparse.RawTextHelpFormatter,
    description="""
    description
        not
           wrapped""",
    epilog="""
    epilog
      not
         wrapped""",
)

parser.add_argument(
    '-a', action="store_true",
    help="""argument
    help is not
    wrapped
    """,
)

parser.print_help()
```

The help text for the `-a` argument is no longer wrapped neatly.

```
$ python3 argparse_raw_text_help_formatter.py

usage: argparse_raw_text_help_formatter.py [-h] [-a]

    description
        not
           wrapped

optional arguments:
  -h, --help  show this help message and exit
  -a          argument
                  help is not
                  wrapped


    epilog
      not
         wrapped
```

Raw formatters may be useful for applications with examples in the description or epilog, where changing the format of the text may make the examples invalid.

The `MetavarTypeHelpFormatter` prints the name of the type for each option, instead of the destination variable, which can be useful for applications with a lot of options of different types.

```python
# argparse_metavar_type_help_formatter.py

import argparse

parser = argparse.ArgumentParser(
    add_help=True,
    formatter_class=argparse.MetavarTypeHelpFormatter,
)

parser.add_argument('-i', type=int, dest='notshown1')
parser.add_argument('-f', type=float, dest='notshown2')

parser.print_help()
```

Rather than display the value of `dest`, the name of the type associated with the option is printed.

```
$ python3 argparse_metavar_type_help_formatter.py

usage: argparse_metavar_type_help_formatter.py [-h] [-i int] [-f
 float]

optional arguments:
  -h, --help  show this help message and exit
  -i int
  -f float
```

Parser Organization

`argparse` includes several features for organizing argument parsers, to make implementation easier or to improve the usability of the help output.

Sharing Parser Rules

Programmers commonly need to implement a suite of command-line tools that all take a set of arguments, and then specialize in some way. For example, if the programs all need to authenticate the user before taking any real action, they would all need to support `--user` and `--password` options. Rather than add the options explicitly to every `ArgumentParser`, it is possible to define a parent parser with the shared options, and then have the parsers for the individual programs inherit from its options.

The first step is to set up the parser with the shared argument definitions. Since each subsequent user of the parent parser is going to try to add the same help options, causing an exception, automatic help generation is turned off in the base parser.

```python
# argparse_parent_base.py

import argparse

parser = argparse.ArgumentParser(add_help=False)

parser.add_argument('--user', action="store")
parser.add_argument('--password', action="store")
```

Next, create another parser with `parents` set:

```python
# argparse_uses_parent.py

import argparse
import argparse_parent_base

parser = argparse.ArgumentParser(
    parents=[argparse_parent_base.parser],
)

parser.add_argument('--local-arg',
                    action="store_true",
                    default=False)

print(parser.parse_args())
```

And the resulting program takes all three options:

And the resulting program takes all three options:

```
$ python3 argparse_uses_parent.py -h

usage: argparse_uses_parent.py [-h] [--user USER]
                               [--password PASSWORD]
                               [--local-arg]

optional arguments:
  -h, --help           show this help message and exit
  --user USER
  --password PASSWORD
  --local-arg
```

Conflicting Options

The previous example pointed out that adding two argument handlers to a parser using the same argument name causes an exception. The conflict resolution behavior can be changed by passing a `conflict_handler`. The two built-in handlers are `error` (the default), and `resolve`, which picks a handler based on the order they are added.

```
# argparse_conflict_handler_resolve.py

import argparse

parser = argparse.ArgumentParser(conflict_handler='resolve')

parser.add_argument('-a', action="store")
parser.add_argument('-b', action="store", help='Short alone')
parser.add_argument('--long-b', '-b',
                    action="store",
                    help='Long and short together')

print(parser.parse_args(['-h']))
```

Since the last handler with a given argument name is used, in this example, the stand-alone option `-b` is masked by the alias for `--long-b`.

```
$ python3 argparse_conflict_handler_resolve.py

usage: argparse_conflict_handler_resolve.py [-h] [-a A]
[--long-b LONG_B]

optional arguments:
  -h, --help            show this help message and exit
  -a A
  --long-b LONG_B, -b LONG_B
                        Long and short together
```

Switching the order of the calls to `add_argument()` unmasks the stand-alone option:

```
# argparse_conflict_handler_resolve2.py

import argparse

parser = argparse.ArgumentParser(conflict_handler='resolve')

parser.add_argument('-a', action="store")
parser.add_argument('--long-b', '-b',
                    action="store",
                    help='Long and short together')
parser.add_argument('-b', action="store", help='Short alone')

print(parser.parse_args(['-h']))
```

Now both options can be used together.

```
$ python3 argparse_conflict_handler_resolve2.py

usage: argparse_conflict_handler_resolve2.py [-h] [-a A]
```

```
                                             [--long-b LONG_B]
                                             [-b B]

optional arguments:
  -h, --help        show this help message and exit
  -a A
  --long-b LONG_B  Long and short together
  -b B              Short alone
```

Argument Groups

argparse combines the argument definitions into “groups.” By default, it uses two groups, with one for options and another for required position-based arguments.

```
# argparse_default_grouping.py

import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('--optional', action="store_true",
                    default=False)
parser.add_argument('positional', action="store")

print(parser.parse_args())
```

The grouping is reflected in the separate “positional arguments” and “optional arguments” sections of the help output.

```
$ python3 argparse_default_grouping.py -h

usage: argparse_default_grouping.py [-h] [--optional] positional

Short sample app

positional arguments:
  positional

optional arguments:
  -h, --help  show this help message and exit
  --optional
```

The grouping can be adjusted to make it more logical in the help, so that related options or values are documented together. The shared-option example from earlier could be written using custom grouping so that the authentication options are shown together in the help.

Create the “authentication” group with `add_argument_group()` and then add each of the authentication-related options to the group, instead of the base parser.

```
# argparse_parent_with_group.py

import argparse

parser = argparse.ArgumentParser(add_help=False)

group = parser.add_argument_group('authentication')

group.add_argument('--user', action="store")
group.add_argument('--password', action="store")
```

The program using the group-based parent lists it in the `parents` value, just as before.

```
# argparse_uses_parent_with_group.py

import argparse
import argparse_parent_with_group

parser = argparse.ArgumentParser(
    parents=[argparse_parent_with_group.parser],
)

parser.add_argument('--local-arg',
                    action="store_true",
                    default=False)

print(parser.parse_args())
```

The help output now shows the authentication options together.

```
$ python3 argparse_uses_parent_with_group.py -h

usage: argparse_uses_parent_with_group.py [-h] [--user USER]
                                          [--password PASSWORD]
                                          [--local-arg]

optional arguments:
  -h, --help           show this help message and exit
  --local-arg

authentication:
  --user USER
  --password PASSWORD
```

Mutually Exclusive Options

Defining mutually exclusive options is a special case of the option grouping feature, and uses `add_mutually_exclusive_group()` instead of `add_argument_group()`.

```
# argparse_mutually_exclusive.py

import argparse

parser = argparse.ArgumentParser()

group = parser.add_mutually_exclusive_group()
group.add_argument('-a', action='store_true')
group.add_argument('-b', action='store_true')

print(parser.parse_args())
```

argparse enforces the mutual exclusivity, so that only one of the options from the group can be given.

```
$ python3 argparse_mutually_exclusive.py -h

usage: argparse_mutually_exclusive.py [-h] [-a | -b]

optional arguments:
  -h, --help  show this help message and exit
  -a
  -b

$ python3 argparse_mutually_exclusive.py -a

Namespace(a=True, b=False)

$ python3 argparse_mutually_exclusive.py -b

Namespace(a=False, b=True)

$ python3 argparse_mutually_exclusive.py -a -b

usage: argparse_mutually_exclusive.py [-h] [-a | -b]
argparse_mutually_exclusive.py: error: argument -b: not allowed
with argument -a
```

Nesting Parsers

The parent parser approach described earlier is one way to share options between related commands. An alternate approach is to combine the commands into a single program, and use sub-parsers to handle each portion of the command-line. The result works in the way svn, hg, and other programs with multiple command-line actions, or sub-commands, does.

A program to work with directories on the file system might define commands for creating, deleting, and listing the contents of a directory.

```
# argparse_subparsers.py
```

```
import argparse

parser = argparse.ArgumentParser()

subparsers = parser.add_subparsers(help='commands')

# A list command
list_parser = subparsers.add_parser(
    'list', help='List contents')
list_parser.add_argument(
    'dirname', action='store',
    help='Directory to list')

# A create command
create_parser = subparsers.add_parser(
    'create', help='Create a directory')
create_parser.add_argument(
    'dirname', action='store',
    help='New directory to create')
create_parser.add_argument(
    '--read-only', default=False, action='store_true',
    help='Set permissions to prevent writing to the directory',
)

# A delete command
delete_parser = subparsers.add_parser(
    'delete', help='Remove a directory')
delete_parser.add_argument(
    'dirname', action='store', help='The directory to remove')
delete_parser.add_argument(
    '--recursive', '-r', default=False, action='store_true',
    help='Remove the contents of the directory, too',
)

print(parser.parse_args())
```

The help output shows the named sub-parsers as “commands” that can be specified on the command-line as positional arguments.

```
$ python3 argparse_subparsers.py -h

usage: argparse_subparsers.py [-h] {list,create,delete} ...

positional arguments:
  {list,create,delete}  commands
    list                List contents
    create              Create a directory
    delete              Remove a directory

optional arguments:
  -h, --help            show this help message and exit
```

Each sub-parser also has its own help, describing the arguments and options for that command.

```
$ python3 argparse_subparsers.py create -h

usage: argparse_subparsers.py create [-h] [--read-only] dirname

positional arguments:
  dirname      New directory to create

optional arguments:
  -h, --help   show this help message and exit
  -h, --help   show this help message and exit
  --read-only  Set permissions to prevent writing to the directo
ry
```

And when the arguments are parsed, the Namespace object returned by `parse_args()` includes only the values related to the command specified.

```
$ python3 argparse_subparsers.py delete -r foo

Namespace(dirname='foo', recursive=True)
```

Advanced Argument Processing

The examples so far have shown simple Boolean flags, options with string or numerical arguments, and positional arguments. `argparse` also supports sophisticated argument specification for variable-length argument list, enumerations, and constant values.

Variable Argument Lists

A single argument definition can be configured to consume multiple arguments on the command-line being parsed. Set `nargs` to one of the flag values from the table below, based on the number of required or expected arguments.

Flags for Variable Argument Definitions in argparse

Value	Meaning
N	The absolute number of arguments (e.g., 3).
?	0 or 1 arguments
*	0 or all arguments
+	All, and at least one, argument

```python
# argparse_nargs.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument('--three', nargs=3)
parser.add_argument('--optional', nargs='?')
parser.add_argument('--all', nargs='*', dest='all')
parser.add_argument('--one-or-more', nargs='+')

print(parser.parse_args())
```

The parser enforces the argument count instructions and generates an accurate syntax diagram as part of the command help text.

```
$ python3 argparse_nargs.py -h

usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                [--optional [OPTIONAL]]
                [--all [ALL [ALL ...]]]
                [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]

optional arguments:
  -h, --help            show this help message and exit
  --three THREE THREE THREE
  --optional [OPTIONAL]
  --all [ALL [ALL ...]]
  --one-or-more ONE_OR_MORE [ONE_OR_MORE ...]

$ python3 argparse_nargs.py

Namespace(all=None, one_or_more=None, optional=None, three=None)

$ python3 argparse_nargs.py --three
```

```
usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                [--optional [OPTIONAL]]
                [--all [ALL [ALL ...]]]
                [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
argparse_nargs.py: error: argument --three: expected 3
argument(s)

$ python3 argparse_nargs.py --three a b c

Namespace(all=None, one_or_more=None, optional=None,
three=['a', 'b', 'c'])

$ python3 argparse_nargs.py --optional

Namespace(all=None, one_or_more=None, optional=None, three=None)

$ python3 argparse_nargs.py --optional with_value

Namespace(all=None, one_or_more=None, optional='with_value',
three=None)

$ python3 argparse_nargs.py --all with multiple values

Namespace(all=['with', 'multiple', 'values'], one_or_more=None,
optional=None, three=None)

$ python3 argparse_nargs.py --one-or-more with_value

Namespace(all=None, one_or_more=['with_value'], optional=None,
three=None)

$ python3 argparse_nargs.py --one-or-more with multiple values

Namespace(all=None, one_or_more=['with', 'multiple', 'values'],
optional=None, three=None)

$ python3 argparse_nargs.py --one-or-more

usage: argparse_nargs.py [-h] [--three THREE THREE THREE]
                [--optional [OPTIONAL]]
                [--all [ALL [ALL ...]]]
                [--one-or-more ONE_OR_MORE [ONE_OR_MORE ...]]
argparse_nargs.py: error: argument --one-or-more: expected
at least one argument
```

Argument Types

argparse treats all argument values as strings, unless it is told to convert the string to another type. The type parameter to add_argument() defines a converter function, which is used by the ArgumentParser to transform the argument value from a string to some other type.

```
# argparse_type.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-i', type=int)
parser.add_argument('-f', type=float)
parser.add_argument('--file', type=open)

try:
    print(parser.parse_args())
except IOError as msg:
    parser.error(str(msg))
```

Any callable that takes a single string argument can be passed as type, including built-in types like int and float or even open().

```
$ python3 argparse_type.py -i 1
$ python3 argparse_type.py -i 1

Namespace(f=None, file=None, i=1)

$ python3 argparse_type.py -f 3.14

Namespace(f=3.14, file=None, i=None)

$ python3 argparse_type.py --file argparse_type.py

Namespace(f=None, file=<_io.TextIOWrapper
name='argparse_type.py' mode='r' encoding='UTF-8'>, i=None)
```

If the type conversion fails, argparse raises an exception. TypeError and ValueError exceptions are trapped automatically and converted to a simple error message for the user. Other exceptions, such as the IOError in the next example where the input file does not exist, must be handled by the caller.

```
$ python3 argparse_type.py -i a

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: argument -i: invalid int value: 'a'

$ python3 argparse_type.py -f 3.14.15

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: argument -f: invalid float value:
'3.14.15'

$ python3 argparse_type.py --file does_not_exist.txt

usage: argparse_type.py [-h] [-i I] [-f F] [--file FILE]
argparse_type.py: error: [Errno 2] No such file or directory:
'does_not_exist.txt'
```

To limit an input argument to a value within a pre-defined set, use the choices parameter.

```
# argparse_choices.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument(
    '--mode',
    choices=('read-only', 'read-write'),
)

print(parser.parse_args())
```

If the argument to --mode is not one of the allowed values, an error is generated and processing stops.

```
$ python3 argparse_choices.py -h

usage: argparse_choices.py [-h] [--mode {read-only,read-write}]

optional arguments:
  -h, --help            show this help message and exit
  --mode {read-only,read-write}

$ python3 argparse_choices.py --mode read-only

Namespace(mode='read-only')

$ python3 argparse_choices.py --mode invalid

usage: argparse_choices.py [-h] [--mode {read-only,read-write}]
argparse_choices.py: error: argument --mode: invalid choice:
'invalid' (choose from 'read-only', 'read-write')
```

File Arguments

Although `file` objects can be instantiated with a single string argument, that does not include the access mode argument. `FileType` provides a more flexible way of specifying that an argument should be a file, including the mode and buffer size.

```
# argparse_FileType.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument('-i', metavar='in-file',
                    type=argparse.FileType('rt'))
parser.add_argument('-o', metavar='out-file',
                    type=argparse.FileType('wt'))

try:
    results = parser.parse_args()
    print('Input file:', results.i)
    print('Output file:', results.o)
except IOError as msg:
    parser.error(str(msg))
```

The value associated with the argument name is the open file handle. The application is responsible for closing the file when it is no longer being used.

```
$ python3 argparse_FileType.py -h

usage: argparse_FileType.py [-h] [-i in-file] [-o out-file]

optional arguments:
  -h, --help   show this help message and exit
  -i in-file
  -o out-file

$ python3 argparse_FileType.py -i argparse_FileType.py -o tmp_\
file.txt

Input file: <_io.TextIOWrapper name='argparse_FileType.py'
mode='rt' encoding='UTF-8'>
Output file: <_io.TextIOWrapper name='tmp_file.txt' mode='wt'
encoding='UTF-8'>

$ python3 argparse_FileType.py -i no_such_file.txt

usage: argparse_FileType.py [-h] [-i in-file] [-o out-file]
argparse_FileType.py: error: argument -i: can't open
'no_such_file.txt': [Errno 2] No such file or directory:
'no_such_file.txt'
```

Custom Actions

In addition to the built-in actions described earlier, custom actions can be defined by providing an object that implements the Action API. The object passed to `add_argument()` as `action` should take parameters describing the argument being defined (all the same arguments given to `add_argument()`) and return a callable object that takes as parameters the `parser` processing the arguments, the `namespace` holding the parse results, the `value` of the argument being acted on, and the `option_string` that triggered the action.

A class `Action` is provided as a convenient starting point for defining new actions. The constructor handles the argument definitions, so only `__call__()` needs to be overridden in the subclass.

```
# argparse_custom_action.py

import argparse


class CustomAction(argparse.Action):
    def __init__(self,
                 option_strings,
                 dest,
                 nargs=None,
                 const=None,
                 default=None,
                 type=None,
                 choices=None,
                 required=False,
                 help=None,
                 metavar=None):
        argparse.Action.__init__(self,
                                 option_strings=option_strings,
                                 dest=dest,
                                 nargs=nargs,
                                 const=const,
                                 default=default,
                                 type=type,
                                 choices=choices,
                                 required=required,
                                 help=help,
                                 metavar=metavar,
                                 )
        print('Initializing CustomAction')
        for name, value in sorted(locals().items()):
            if name == 'self' or value is None:
                continue
            print('  {} = {!r}'.format(name, value))
        print()
        return

    def __call__(self, parser, namespace, values,
                 option_string=None):
        print('Processing CustomAction for {}'.format(self.dest))
        print('  parser = {}'.format(id(parser)))
        print('  values = {!r}'.format(values))
        print('  option_string = {!r}'.format(option_string))

        # Do some arbitrary processing of the input values
        if isinstance(values, list):
            values = [v.upper() for v in values]
        else:
            values = values.upper()
        # Save the results in the namespace using the destination
        # variable given to our constructor.
        setattr(namespace, self.dest, values)
        print()


parser = argparse.ArgumentParser()

parser.add_argument('-a', action=CustomAction)
parser.add_argument('-m', nargs='*', action=CustomAction)

results = parser.parse_args(['-a', 'value',
                             '-m', 'multivalue',
                             'second'])
print(results)
```

The type of values depends on the value of nargs. If the argument allows multiple values, values will be a list even if it only contains one item.

The value of option_string also depends on the original argument specification. For positional required arguments, option_string is always None.

```
$ python3 argparse_custom_action.py

Initializing CustomAction
  dest = 'a'
  option_strings = ['-a']
  required = False

Initializing CustomAction
  dest = 'm'
  nargs = '*'
  option_strings = ['-m']
  required = False

Processing CustomAction for a
  parser = 4315836992
  values = 'value'
  option_string = '-a'

Processing CustomAction for m
  parser = 4315836992
  values = ['multivalue', 'second']
  option_string = '-m'

Namespace(a='VALUE', m=['MULTIVALUE', 'SECOND'])
```

See also

- Standard library documentation for argparse
- `configparser` – Read and write configuration files.
- `shlex` – Parse shell-like syntaxes.
- Python 2 to 3 porting notes for argparse

Application Building Blocks

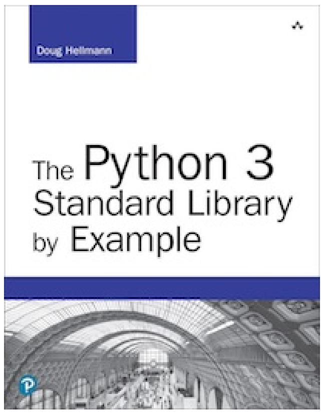
getopt — Command Line Option Parsing

Quick Links

This page was last updated 2016-12-30.

Navigation

Application Building Blocks
getopt — Command Line Option Parsing



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?