`!optparse` --- Parser for command line options

- Choosing an argument parsing library
- Introduction
- Background
  - Terminology
  - What are options for?
  - What are positional arguments for?
- Tutorial
  - Understanding option actions
  - The store action
  - Handling boolean (flag) options
  - Other actions
  - Default values
  - Generating help
    - Grouping Options
  - Printing a version string
  - How !optparse handles errors
  - Putting it all together
- Reference Guide
  - Creating the parser
  - Populating the parser
  - Defining options
  - Option attributes
  - Standard option actions
  - Standard option types
  - Parsing arguments
  - Querying and manipulating your option parser

- Conflicts between options
- Cleanup
- Other methods
- Option Callbacks
  - Defining a callback option
  - How callbacks are called
  - Raising errors in a callback
  - Callback example 1: trivial callback
  - Callback example 2: check option order
  - Callback example 3: check option order (generalized)
  - Callback example 4: check arbitrary condition
  - Callback example 5: fixed arguments
  - Callback example 6: variable arguments
- Extending !optparse
  - Adding new types
  - Adding new actions
- Exceptions

---

optparse

Source code: `Lib/optparse.py`

Choosing an argument parsing library

The standard library includes three argument parsing libraries:

- `getopt`: a module that closely mirrors the procedural C `getopt` API.
- `!optparse`: a declarative replacement for `getopt` that provides equivalent functionality without requiring each application to implement its own procedural option parsing logic.
- `argparse`: a more opinionated alternative to `optparse` that provides more functionality by default, at the expense of reduced application flexibility in controlling exactly how arguments are processed.

In the absence of more specific argument parsing design constraints, `argparse` is the recommended choice for implementing command line applications, as it offers the highest level of baseline functionality with the least application level code.

`getopt` is retained almost entirely for backwards compatibility reasons. However, it also serves a niche use case as a tool for prototyping and testing command line argument handling in `getopt`-based C applications.

`!optparse` should be considered as an alternative to `argparse` in the following cases:

- an application is already using `!optparse` and doesn't want to risk the subtle behavioural changes that may arise when migrating to `argparse`
- the application requires additional control over the way options and positional parameters are interleaved on the command line (including the ability to disable the interleaving feature completely)
- the application requires additional control over the incremental parsing of command line elements (while `argparse` does support this, the exact way it works in practice is undesirable for some use cases)
- the application requires additional control over the handling of options which accept parameter values that may start with `-` (such as delegated options to be passed to invoked subprocesses)
- the application requires some other command line parameter processing behavior which `argparse` does not support, but which can be implemented in terms of the lower level interface offered by `optparse`

These considerations also mean that `!optparse` is likely to provide a better foundation for library authors writing third party command line argument processing libraries.

As a concrete example, consider the following two command line argument parsing configurations, the first using `optparse`, and the second using `argparse`:

import optparse

if __name__ == '__main__':
parser = optparse.OptionParser() parser.add_option('-o', '--output') parser.add_option('-v', dest='verbose', action='store_true') opts, args = parser.parse_args() process(args, output=opts.output, verbose=opts.verbose)

import argparse

if __name__ == '__main__':
parser = argparse.ArgumentParser() parser.add_argument('-o', '--output') parser.add_argument('-v', dest='verbose', action='store_true') parser.add_argument('rest', nargs='*') args = parser.parse_args() process(args.rest, output=args.output, verbose=args.verbose)

The most obvious difference is that in the `optparse` version, the non-option arguments are processed separately by the application after the option processing is complete. In the `argparse` version, positional arguments are declared and processed in the same way as the named options.

However, the `argparse` version will also handle some parameter combination differently from the way the `optparse` version would handle them. For example (amongst other differences):

- supplying `-o -v` gives `output="-v"` and `verbose=False` when using `optparse`, but a usage error with `argparse` (complaining that no value has been supplied for `-o/--output`, since `-v` is interpreted as meaning the verbosity flag)
- similarly, supplying `-o --` gives `output="--"` and `args=()` when using `optparse`, but a usage error with `argparse` (also complaining that no value has been supplied for `-o/--output`, since `--` is interpreted as terminating the option processing and treating all remaining values as positional arguments)
- supplying `-o=foo` gives `output="=foo"` when using `optparse`, but gives `output="foo"` with `argparse` (since `=` is special cased as an alternative separator for option parameter values)

Whether these differing behaviors in the `argparse` version are considered desirable or a problem will depend on the specific command line application use case.

`click` is a third party argument processing library (originally based on `optparse`), which allows command line applications to be developed as a set of decorated command implementation functions.

Other third party libraries, such as `typer` or `msgspec-click`, allow command line interfaces to be specified in ways that more effectively integrate with static checking of Python type annotations.

Introduction

`!optparse` is a more convenient, flexible, and powerful library for parsing command-line options than the minimalist `getopt` module. `!optparse` uses a more declarative style of command-line parsing: you create an instance of `OptionParser`, populate it with options, and parse the command line. `!optparse` allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using `!optparse` in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
```

```
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, `!optparse` sets attributes of the `options` object returned by `~OptionParser.parse_args` based on user-supplied command-line values. When `~OptionParser.parse_args` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. `!optparse` supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following :

```
<yourscript> -h
<yourscript> --help
```

and `!optparse` will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

Background

`!optparse` was explicitly designed to encourage the creation of programs with straightforward command-line interfaces that follow the conventions established by the `!getopt` family of functions available to C developers. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, reading this section will allow you to acquaint yourself with them.

Terminology

argument
a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

option
an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by `!optparse`.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by `!optparse`, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

option argument
an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With `!optparse`, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, `!optparse` does not support this feature.

positional argument
something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option
an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. `!optparse` doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`---all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user---most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply---use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options---the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

Tutorial

While `!optparse` is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any `!optparse`-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell `!optparse` what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to `OptionParser.add_option` are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, `!optparse` encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct `!optparse` to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `~OptionParser.parse_args`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`~OptionParser.parse_args` returns two values:

- `options`, an object containing values for all of your options---e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: `~Option.action`, `~Option.type`, `~Option.dest` (destination), and `~Option.help`. Of these, `~Option.action` is the most fundamental.

Understanding option actions

Actions tell `!optparse` what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into `!optparse`; adding new actions is an advanced topic covered in section `optparse-extending-optparse`. Most actions tell `!optparse` to store a value in some variable---for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, `!optparse` defaults to `store`.

The store action

The most common option action is `store`, which tells `!optparse` to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask `!optparse` to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When `!optparse` sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `~OptionParser.parse_args`, `options.filename` is `"foo.txt"`.

Some other option types supported by `!optparse` are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code :

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print `42`.

If you don't specify a type, `!optparse` assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, `!optparse` figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, `!optparse` looks at the first short option string: the default destination for `-f` is `f`.

`!optparse` also includes the built-in `complex` type. Adding types is covered in section `optparse-extending-optparse`.

Handling boolean (flag) options

Flag options---set a variable to true or false when a particular option is seen---are quite common. `!optparse` supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values---see below.)

When `!optparse` encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by `!optparse` are:

`"store_const"`
store a constant value, pre-set via `Option.const`

`"append"`
append this option's argument to a list

`"count"`
increment a counter by one

`"callback"`
call a specified function

These are covered in section `optparse-reference-guide`, and section `optparse-option-callbacks`.

Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. `!optparse` lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want `!optparse` to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults` method of OptionParser, which you can call at any time before calling `~OptionParser.parse_args`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

`!optparse`'s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a `~Option.help` value for each option, and

optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If `!optparse` encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, `!optparse` exits after printing the help text.)

There's a lot going on here to help `!optparse` generate the best possible help message:

- the script defines its own usage message:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  `!optparse` expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don't supply a usage string, `!optparse` uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping---`!optparse` takes care of wrapping lines and making the help output look good.
- options that take a value indicate this fact in their automatically generated help message, e.g. for the "mode" option:

```
-m MODE, --mode=MODE
```

Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, `!optparse` converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want---for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically generated option description:

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string---`!optparse` will replace it with `str` of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An `OptionParser` can contain several option groups, each of which can contain several options.

An option group is obtained using the class `OptionGroup`:

where

- parser is the `OptionParser` instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

`OptionGroup` inherits from `OptionContainer` (like `OptionParser`) and so the `add_option` method can be used to add an option to the group.

Once all the options are declared, using the `OptionParser` method `add_option_group` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an `OptionGroup` to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
```

```
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

OptionParser.get_option_group(opt_str)

Return the `OptionGroup` to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such `OptionGroup`, return `None`.

Printing a version string

Similar to the brief usage string, `!optparse` can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, `!optparse` automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

OptionParser.print_version(file=None)

Print the version message for the current program (`self.version`) to *file* (default stdout). As with `print_usage`, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

OptionParser.get_version()

Same as `print_version` but returns the version string instead of printing it.

How `!optparse` handles errors

There are two broad classes of errors that `!optparse` has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to `OptionParser.add_option`, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or `TypeError`) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. `!optparse` can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, `!optparse` handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

`!optparse`-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error` from your application code.

If `!optparse`'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `~OptionParser.exit` and/or `~OptionParser.error` methods.

Putting it all together

Here's what `!optparse`-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
```

```
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

Reference Guide

Creating the parser

The first step in using `!optparse` is to create an OptionParser instance.

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

`usage` (default: `"%prog [options]"`)
The usage summary to print when your program is run incorrectly or with a help option. When `!optparse` prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

`option_list` (default: `[]`)
A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use `add_option` after creating the parser instead.

`option_class` (default: optparse.Option)
Class to use when adding options to the parser in `add_option`.

`version` (default: `None`)
A version string to print when the user supplies a version option. If you supply a true value for `version`, `!optparse` automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

`conflict_handler` (default: `"error"`)
Specifies what to do when options with conflicting option strings are added to the parser; see section `optparse-conflicts-between-options`.

`description` (default: `None`)
A paragraph of text giving a brief overview of your program. `!optparse` reformats this paragraph to

fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

`formatter` (default: a new `IndentedHelpFormatter`)
An instance of optparse.HelpFormatter that will be used for printing help text. `!optparse` provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

`add_help_option` (default: `True`)
If true, `!optparse` will add a help option (with option strings `-h` and `--help`) to the parser.

`prog`
The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

`epilog` (default: `None`)
A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using `OptionParser.add_option`, as shown in section `optparse-tutorial`. `add_option` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of `!optparse` may split Option into several classes, and `make_option` will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option` method of `OptionParser`.

OptionParser.add_option(option) OptionParser.add_option(*opt_str, attr=value, ...)

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is `~Option.action`, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, `!optparse` raises an `OptionError` exception explaining your mistake.

An option's *action* determines what `!optparse` does when it encounters this option on the command-line. The standard option actions hard-coded into `!optparse` are:

`"store"`
store this option's argument (default)

`"store_const"`
store a constant value, pre-set via `Option.const`

`"store_true"`
store `True`

`"store_false"`
store `False`

`"append"`
append this option's argument to a list

`"append_const"`
append a constant value to a list, pre-set via `Option.const`

`"count"`
increment a counter by one

`"callback"`
call a specified function

`"help"`
print a usage message including all options and the documentation for them

(If you don't supply an action, the default is `"store"`. For this action, you may also supply `~Option.type` and `~Option.dest` option attributes; see `optparse-standard-option-actions`.)

As you can see, most actions involve storing or updating a value somewhere. `!optparse` always creates a special object for this, conventionally called `options`, which is an instance of `optparse.Values`.

An object holding parsed argument names and values as attributes. Normally created by calling when calling `OptionParser.parse_args`, and can be overridden by a custom subclass passed to the *values* argument of `OptionParser.parse_args` (as described in `optparse-parsing-arguments`).

`!Values` objects support `copy.replace`, which returns a copy of the object with the specified attributes replaced.

3.16 Added support for `copy.replace`.

Option arguments (and various other values) are stored as attributes of this object, according to the `~Option.dest` (destination) option attribute.

For example, when you call :

```
parser.parse_args()
```

one of the first things `!optparse` does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with :

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then `!optparse`, on seeing this option, will do the equivalent of :

```
options.filename = "foo"
```

The `~Option.type` and `~Option.dest` option attributes are almost as important as `~Option.action`, but `~Option.action` is the only one that makes sense for *all* options.

Option attributes

A single command line argument, with various attributes passed by keyword to the constructor. Normally created with `OptionParser.add_option` rather than directly, and can be overridden by a custom class via the *option_class* argument to `OptionParser`.

The following option attributes may be passed as keyword arguments to `OptionParser.add_option`. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, `!optparse` raises `OptionError`.

Option.action

(default: `"store"`)

Determines `!optparse`'s behaviour when this option is seen on the command line; the available options are documented `here <optparse-standard-option-actions>`.

Option.type

(default: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented `here <optparse-standard-option-types>`.

Option.dest

(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells `!optparse` where to write it: `~Option.dest` names an attribute of the `options` object that `!optparse` builds as it parses the command line.

Option.default

The value to use for this option's destination if the option is not seen on the command line. See also `OptionParser.set_defaults`.

Option.nargs

(default: 1)

How many arguments of type `~Option.type` should be consumed when this option is seen. If > 1, `!optparse` will store a tuple of values to `~Option.dest`.

Option.const

For actions that store a constant value, the constant value to store.

Option.choices

For options of type `"choice"`, the list of strings the user may choose from.

Option.callback

For options with action `"callback"`, the callable to call when this option is seen. See section `optparse-option-callbacks` for detail on the arguments passed to the callable.

Option.callback_args Option.callback_kwargs

Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

Option.help

Help text to print for this option when listing all available options after the user supplies a `~Option.help` option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option.metavar

(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section `optparse-tutorial` for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide `!optparse`'s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: `~Option.type`, `~Option.dest`, `~Option.nargs`, `~Option.choices`]

The option must be followed by an argument, which is converted to a value according to `~Option.type` and stored in `~Option.dest`. If `~Option.nargs` > 1, multiple arguments will be consumed from the command line; all will be converted according to `~Option.type` and stored to `~Option.dest` as a tuple. See the `optparse-standard-option-types` section.
If `~Option.choices` is supplied (a list or tuple of strings), the type defaults to `"choice"`.
If `~Option.type` is not supplied, it defaults to `"string"`.
If `~Option.dest` is not supplied, `!optparse` derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, `!optparse` derives a destination from the first short option string (e.g., `-f` implies `f`).
Example:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

As it parses the command line :

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

`!optparse` will set :

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- `"store_const"` [required: `~Option.const`; relevant: `~Option.dest`]
  The value `~Option.const` is stored in `~Option.dest`.
  Example:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If `--noisy` is seen, `!optparse` will set :

```
options.verbose = 2
```

- `"store_true"` [relevant: `~Option.dest`]
  A special case of `"store_const"` that stores `True` to `~Option.dest`.
- `"store_false"` [relevant: `~Option.dest`]
  Like `"store_true"`, but stores `False`.
  Example:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- `"append"` [relevant: `~Option.type`, `~Option.dest`, `~Option.nargs`, `~Option.choices`]
  The option must be followed by an argument, which is appended to the list in `~Option.dest`. If no default value for `~Option.dest` is supplied, an empty list is automatically created when `!optparse` first encounters this option on the command-line. If `~Option.nargs` > 1, multiple arguments are consumed, and a tuple of length `~Option.nargs` is appended to `~Option.dest`.
  The defaults for `~Option.type` and `~Option.dest` are the same as for the `"store"` action.
  Example:

  ```
  parser.add_option("-t", "--tracks", action="append", type="int")
  ```

  If `-t3` is seen on the command-line, `!optparse` does the equivalent of:

  ```
  options.tracks = []
  options.tracks.append(int("3"))
  ```

  If, a little later on, `--tracks=4` is seen, it does:

  ```
  options.tracks.append(int("4"))
  ```

  The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

  ```
  >>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
  >>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
  >>> opts.files
  ['~/.mypkg/defaults', 'overrides.mypkg']
  ```

- `"append_const"` [required: `~Option.const`; relevant: `~Option.dest`]
  Like `"store_const"`, but the value `~Option.const` is appended to `~Option.dest`; as with `"append"`, `~Option.dest` defaults to `None`, and an empty list is automatically created the first time the option is encountered.
- `"count"` [relevant: `~Option.dest`]
  Increment the integer stored at `~Option.dest`. If no default value is supplied, `~Option.dest` is set to zero before being incremented the first time.
  Example:

  ```
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  The first time `-v` is seen on the command line, `!optparse` does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in :

```
options.verbosity += 1
```

- `"callback"` [required: `~Option.callback`; relevant: `~Option.type`, `~Option.nargs`, `~Option.callback_args`, `~Option.callback_kwargs`]
  Call the function specified by `~Option.callback`, which is called as :

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  See section `optparse-option-callbacks` for more detail.
- `"help"`
  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the `~Option.help` string passed to every option.
  If no `~Option.help` string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.
  `!optparse` automatically adds a `~Option.help` option to all OptionParsers, so you do not normally need to create one.
  Example:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  If `!optparse` sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
```

```
-v                Be moderately verbose
--file=FILENAME   Input file to read data from
```

After printing the help message, `!optparse` terminates your process with `sys.exit(0)`.

- `"version"`
  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with `~Option.help` options, you will rarely create `version` options, since `!optparse` automatically adds them when needed.

Standard option types

`!optparse` has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section `optparse-extending-optparse`.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling `int` with the appropriate base (2, 8, 10, or 16). If this fails, so will `!optparse`, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with `float` and `complex`, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The `~Option.choices` option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

Parsing arguments

The whole point of creating and populating an OptionParser is to call its `~OptionParser.parse_args` method.

OptionParser.parse_args(args=None, values=None)

Parse the command-line options found in *args*.

The input parameters are

`args`
the list of arguments to process (default: `sys.argv[1:]`)

`values`
a `Values` object to store option arguments in (default: a new instance of `Values`) -- if you give an existing object, the option defaults will not be initialized on it

and the return value is a pair `(options, args)` where

`options`
the same object that was passed in as *values*, or the `optparse.Values` instance created by `!optparse`

`args`
the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated `setattr` calls (roughly one for every option argument stored to an option destination) and returned by `~OptionParser.parse_args`.

If `~OptionParser.parse_args` encounters any errors in the argument list, it calls the OptionParser's `error` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

OptionParser.disable_interspersed_args()

Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, `!optparse` normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to :

```
prog -a -b arg1 arg2
```

To disable this feature, call `disable_interspersed_args`. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

OptionParser.enable_interspersed_args()

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

OptionParser.get_option(opt_str)

Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

OptionParser.has_option(opt_str)

Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

OptionParser.remove_option(opt_str)

If the `OptionParser` has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this `OptionParser`, raises `ValueError`.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, `!optparse` checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

> `"error"` (default)
> assume option conflicts are a programming error and raise `OptionConflictError`
>
> `"resolve"`
> resolve option conflicts intelligently (see below)

As an example, let's define an `OptionParser` that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, `!optparse` detects that a previously added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, `!optparse` removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so `!optparse` removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `~OptionParser.destroy` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

OptionParser.set_usage(usage)

Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

OptionParser.print_usage(file=None)

Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

OptionParser.get_usage()

Same as `print_usage` but returns the usage string instead of printing it.

OptionParser.set_defaults(dest=value, ...)

Set default values for several option destinations at once. Using `set_defaults` is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use `set_defaults`:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

Option Callbacks

When `!optparse`'s built-in actions and types aren't quite enough for your needs, you have two choices: extend `!optparse` or define a callback option. Extending `!optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the `OptionParser.add_option` method. Apart from `~Option.action`, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `!optparse` doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments---the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing

callbacks gets tricky; it's covered later in this section.

`!optparse` always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via `~Option.callback_args` and `~Option.callback_kwargs`. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

`~Option.type`
has its usual meaning: as with the `"store"` or `"append"` actions, it instructs `!optparse` to consume one argument and convert it to `~Option.type`. Rather than storing the converted value(s) anywhere, though, `!optparse` passes it to your callback function.

`~Option.nargs`
also has its usual meaning: if it is supplied and > 1, `!optparse` will consume `~Option.nargs` arguments, each of which must be convertible to `~Option.type`. It then passes a tuple of converted values to your callback.

`~Option.callback_args`
a tuple of extra positional arguments to pass to the callback

`~Option.callback_kwargs`
a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

`option`
is the Option instance that's calling the callback

`opt_str`
is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string---e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

`value`
is the argument to this option seen on the command-line. `!optparse` will only expect an argument if `~Option.type` is set; the type of `value` will be the type implied by the option's type. If `~Option.type` for this option is `None` (no argument expected), then `value` will be `None`. If `~Option.nargs` > 1, `value` will be a tuple of values of the appropriate type.

`parser`
is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

`parser.largs`
the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `~OptionParser.parse_args`.)

`parser.rargs`
the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

`parser.values`
the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of `!optparse` for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

`args`
is a tuple of arbitrary positional arguments supplied via the `~Option.callback_args` option attribute.

`kwargs`
is a dictionary of arbitrary keyword arguments supplied via `~Option.callback_kwargs`.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). `!optparse` catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line. :

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to reuse this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized. :

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there---you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option: if you define `~Option.type`, then the option takes one argument that must be convertible to that type; if you further define `~Option.nargs`, then the option takes `~Option.nargs` arguments.

Here's an example that just emulates the standard `"store"` action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that `!optparse` takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as `!optparse` doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that `!optparse` normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why `!optparse` doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)
```

```
    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

Extending `!optparse`

Since the two major controlling factors in how `!optparse` interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of `!optparse`'s `Option` class. This class has a couple of attributes that define `!optparse`'s types: `~Option.TYPES` and `~Option.TYPE_CHECKER`.

Option.TYPES

A tuple of type names; in your subclass, simply define a new tuple `TYPES` that builds on the standard one.

Option.TYPE_CHECKER

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args`, or be passed to a callback as the `value` parameter.

Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to `OptionParser`'s `error` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because `!optparse` 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the `~Option.TYPE_CHECKER` class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a `~copy.copy` of `Option.TYPE_CHECKER`, we would end up modifying the `~Option.TYPE_CHECKER` attribute of `!optparse`'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other `!optparse`-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that `!optparse` has a couple of classifications for actions:

"store" actions
actions that result in `!optparse` storing a value to an attribute of the current OptionValues instance; these options require a `~Option.dest` attribute to be supplied to the Option constructor.

"typed" actions
actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a `~Option.type` attribute to the Option constructor.

These are overlapping sets: some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

Option.ACTIONS

All actions must be listed in ACTIONS.

Option.STORE_ACTIONS

"store" actions are additionally listed here.

Option.TYPED_ACTIONS

"typed" actions are additionally listed here.

Option.ALWAYS_TYPED_ACTIONS

Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that `!optparse` assigns the default type, `"string"`, to options with no explicit type whose action is listed in `ALWAYS_TYPED_ACTIONS`.

In order to actually implement your new action, you must override Option's `take_action` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line :

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list :

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both `~Option.STORE_ACTIONS` and `~Option.TYPED_ACTIONS`.
- to ensure that `!optparse` assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in `~Option.ALWAYS_TYPED_ACTIONS` as well.
- `MyOption.take_action` implements just this one new action, and passes control back to `Option.take_action` for the standard `!optparse` actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value` method. `ensure_value` is essentially `getattr` with a safety valve; it is called as :

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns `value`. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value`

will take care of getting it right when it's needed.

Exceptions

OptionError

Raised if an `Option` instance is created with invalid or inconsistent arguments.

OptionConflictError

Raised if conflicting options are added to an `OptionParser`.

OptionValueError

Raised if an invalid option value is encountered on the command line.

BadOptionError

Raised if an invalid option is passed on the command line.

AmbiguousOptionError

Raised if an ambiguous option is passed on the command line.

□□□ #2
□ □□□ □□□ 2026-10-31 17:15:57 UTC
□ □□□ □□□ 2026-10-31 17:16:03 UTC