argparse Documentation

Release 1.0.1

Steven Bethard

September 14, 2009

CONTENTS

CHAPTER

ONE

INTRODUCTION TO ARGPARSE

Pretty much every script that uses the argparse module will start out by creating an `ArgumentParser` object. Typically, this will look something like:

```
>>> parser = argparse.ArgumentParser(description='Frabble the foo and the bars')
```

The `ArgumentParser` object will hold all the information necessary to parse the command line into a more manageable form for your program.

1.1 Adding arguments

Once you've created an `ArgumentParser`, you'll want to fill it with information about your program arguments. You typically do this by making calls to the `add_argument()` method. Generally, these calls tell the `ArgumentParser` how to take the strings on the command line and turn them into objects for you. This information is stored and used when `parse_args()` is called. For example, if we add some arguments like this:

```
>>> parser.add_argument('-f', '--foo', action='store_true', help='frabble the foos')
>>> parser.add_argument('bar', nargs='+', type=int, help='a bar to be frabbled')
```

when we later call `parse_args()`, we can expect it to return an object with two attributes, `foo` and `bar`. The `foo` attribute will be `True` if `--foo` was supplied at the command-line, and the `bar` attribute will be a list of ints determined from the remaining command-line arguments:

```
>>> parser.parse_args('--foo 1 2 3 5 8'.split())
Namespace(bar=[1, 2, 3, 5, 8], foo=True)
```

As you can see from the example above, calls to `add_argument()` start with either a single string name for positional arguments or a series of option strings (beginning with `'-'`) for optional arguments. The remaining keyword arguments to `add_argument()` specify exactly what sort of action should be carried out when the `ArgumentParser` object encounters the corresponding command-line args. So in our example above, we are telling the `ArgumentParser` object that when it encounters `--foo` in the command-line args, it should invoke the `'store_true'` action.

1.2 Parsing arguments

Once an `ArgumentParser` has been initialized with appropriate calls to `add_argument()`, it can be instructed to parse the command-line args by calling the `parse_args()` method. This will inspect the command-line, convert

each arg to the appropriate type and then invoke the appropriate action. In most cases, this means a simple namespace object will be built up from attributes parsed out of the command-line.

In the most common case, `parse_args()` will be called with no arguments, and the `ArgumentParser` will determine the command-line args from `sys.argv`. The following example sets up a simple `ArgumentParser` and then calls `parse_args()` in this manner:

```
import argparse

if __name__ == '__main__':
parser = argparse.ArgumentParser()
parser.add_argument(
    'integers', metavar='int', type=int, choices=xrange(10),
     nargs='+', help='an integer in the range 0..9')
parser.add_argument(
    '--sum', dest='accumulate', action='store_const', const=sum,
  default=max, help='sum the integers (default: find the max)')

args = parser.parse_args()
print args.accumulate(args.integers)
```

Assuming this program is saved in the file `script.py`, the call to `parse_args()` means that we get the following behavior when running the program from the command-line:

```
$ script.py 1 2 3 4
4

$ script.py --sum 1 2 3 4
10
```

That's pretty much it. You're now ready to go write some command line interfaces!

CHAPTER

TWO

ARGPARSE VS. OPTPARSE

The optparse module already comes in the Python standard library. So why would you want to use argparse instead? Here's a few of the many reasons:

- The argparse module can handle positional and optional arguments, while optparse can handle only optional arguments. (See `add_argument()`.)
- The argparse module isn't dogmatic about what your command line interface should look like - options like `-file` or `/file` are supported, as are required options. Optparse refuses to support these features, preferring purity over practicality.
- The argparse module produces more informative usage messages, including command-line usage determined from your arguments, and help messages for both positional and optional arguments. The optparse module requires you to write your own usage string, and has no way to display help for positional arguments.
- The argparse module supports action that consume a variable number of command-line args, while optparse requires that the exact number of arguments (e.g. 1, 2, or 3) be known in advance. (See `add_argument()`.)
- The argparse module supports parsers that dispatch to sub-commands, while optparse requires setting `allow_interspersed_args` and doing the parser dispatch manually. (See `add_subparsers()`.)
- The argparse module allows the `type` and `action` parameters to `add_argument()` to be specified with simple callables, while optparse requires hacking class attributes like `STORE_ACTIONS` or `CHECK_METHODS` to get proper argument checking. (See `add_argument()`).

The following sections discuss some of these points and a few of the other advantages of the argparse module.

2.1 Advantages of argparse

2.1.1 Positional arguments

The argparse module supports optional and positional arguments in a manner similar to the optparse module:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', help='optional foo')
>>> parser.add_argument('bar', help='positional bar')
>>> values = parser.parse_args('--foo spam badger'.split())
>>> values.foo
'spam'
>>> values.bar
'badger'
```

Some of the important differences illustrated here:

- ArgumentParser objects use an `add_argument()` method instead of `add_option`. The APIs are quite similar to the optparse ones, and support the keyword arguments `action`, `dest`, `nargs`, `const`, `default`, `type`, `choices`, `help` and `metavar` in much the same way that optparse does (with differences noted below).
- The `parse_args()` method returns a single namespace object, not a `(namespace, remaining_args)` pair. What used to be remaining arguments with optparse are now taken care of by argparse's positional arguments.

2.1.2 Optional arguments

The argparse module doesn't try to dictate what your command line interface should look like. You want single dashes but long option names? Sure! You want to use `'+'` as a flag character? Sure! You want required options? Sure!

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('-foo')
>>> parser.add_argument('+bar', required=True)
>>> parser.parse_args('-foo 1 +bar 2'.split())
Namespace(bar='2', foo='1')
>>> parser.parse_args('-foo X'.split())
usage: PROG [-h] [-foo FOO] +bar BAR
PROG: error: argument +bar is required
```

2.1.3 Generated usage

With optparse, if you don't supply a usage string, you typically end up with `"PROG [options]"` displayed for usage. Since the ArgumentParser objects know both your optional and positional arguments, if you don't supply your own usage string, a reasonable one will be derived for you:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-w', choices='123', help='w help')
>>> parser.add_argument('-x', nargs='+', help='x help')
>>> parser.add_argument('y', nargs='?', help='y help')
>>> parser.add_argument('z', nargs='*', help='z help')
>>> parser.parse_args('-x 2 3 -w 1 a b c d'.split())
Namespace(w='1', x=['2', '3'], y='a', z=['b', 'c', 'd'])
>>> parser.print_help()
usage: PROG [-h] [-w {1,2,3}] [-x X [X ...]] [y] [z [z ...]]

positional arguments:
  y             y help
  z             z help

optional arguments:
  -h, --help    show this help message and exit
  -w {1,2,3}    w help
  -x X [X ...]  x help
```

2.1.4 More nargs options

As you may have noticed in the previous section, the argparse module adds a number of useful new specifiers for the `nargs` keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs='?', const='X')
>>> parser.add_argument('-y', nargs='+')
>>> parser.add_argument('z', nargs='*')
>>> parser.parse_args('-y 0.5 -x'.split())
Namespace(x='X', y=['0.5'], z=[])
>>> parser.parse_args('-y 0.5 -xA 0 1 1 0'.split())
Namespace(x='A', y=['0.5'], z=['0', '1', '1', '0'])
```

In particular argparse supports:

- `N` (an integer) meaning that `N` string args are allowed.
- A `'?'`, meaning that zero or one string args are allowed.
- A `'*'`, meaning that zero or more string args are allowed.
- A `'+'`, meaning that one or more string args are allowed.

By default, a single argument is accepted. For everything but `'?'` and the default, a list of values will be produced instead of single value.

2.1.5 Sub-commands

With optparse, dispatching to subparsers required disallowing interspersed args and then manually matching arg names to parsers. With the argparse module, sub-parsers are supported through the `add_subparsers()` method. The `add_subparsers()` method creates and returns a positional argument that exposes an `add_parser` method from which new named parsers can be created:

```
>>> # create the base parser with a subparsers argument
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--bar')
>>> subparsers = parser.add_subparsers()

>>> # add a sub-command "abc"
>>> parser_abc = subparsers.add_parser('abc')
>>> parser_abc.add_argument('-a', action='store_true')
>>> parser_abc.add_argument('--b', type=int)
>>> parser_abc.add_argument('c', nargs=2)

>>> # add a sub-command "xyz"
>>> parser_xyz = subparsers.add_parser('xyz')
>>> parser_xyz.add_argument('--x', dest='xxx')
>>> parser_xyz.add_argument('-y', action='store_const', const=object)
>>> parser_xyz.add_argument('z', choices='123')

>>> # parse, using the subcommands
>>> parser.parse_args('abc --b 2 AA BB'.split())
Namespace(a=None, b=2, bar=None, c=['AA', 'BB'])
>>> parser.parse_args('--bar B xyz -y 3'.split())
Namespace(bar='B', xxx=None, y=<type 'object'>, z='3')
```

```
>>> parser.parse_args('xyz --b 42'.split())
usage: PROG xyz [-h] [--x XXX] [-y] {1,2,3}
PROG xyz: error: no such option: --b
```

Note that in addition to all the usual arguments that are valid to the `ArgumentParser` constructor, the `add_parser` method of a sub-parsers argument requires a name for the parser. This is used to determine which parser is invoked at argument parsing time, and to print a more informative usage message.

2.1.6 Callable types

The argparse module allows any callable that takes a single string argument as the value for the `type` keyword argument:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('x', type=float)
>>> parser.add_argument('y', type=complex)
>>> parser.add_argument('z', type=file)
>>> parser.parse_args('0.625 4j argparse.py'.split())
Namespace(x=0.625, y=4j, z=<open file 'argparse.py', mode 'r' at 0x...>)
```

For most users, you'll never need to specify a type in string form again.

2.1.7 Extensible actions

The argparse module allows a more easily extensible means of providing new types of parsing actions. The easiest way of generating such a new action is to extend `argparse.Action` and override the `__init__()` and `__call__()` methods as necessary:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, foo, **kwargs):
...         super(FooAction, self).__init__(**kwargs)
...         self.foo = foo
...     def __call__(self, parser, namespace, value, option_string=None):
...         setattr(namespace, self.dest, self.foo % value)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-x', action=FooAction, foo='xfoox(%s)')
>>> parser.add_argument('y', action=FooAction, foo='fyoyo(%s)')
>>> parser.parse_args('42'.split())
Namespace(x=None, y='fyoyo(42)')
>>> parser.parse_args('42 -x 0'.split())
Namespace(x='xfoox(0)', y='fyoyo(42)')
```

The ArgumentParser constructs your action object when `add_argument()` is called, and passes on the arguments it received. Thus if you need more than the usual `dest`, `nargs`, etc., simply declare it in your `__init__()` method and provide a value for it in the corresponding call.

2.1.8 More choices

In optparse, the `choices` keyword argument accepts only a list of strings. The argparse module allows `choices` to provide any container object, and tests the arg string values against this container after they have been type-converted:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs='+', choices='abc')
>>> parser.add_argument('-y', type=int, choices=xrange(3))
>>> parser.add_argument('z', type=float, choices=[0.5, 1.5])
>>> parser.parse_args('-x a c -y 2 0.5'.split())
Namespace(x=['a', 'c'], y=2, z=0.5)
>>> parser.parse_args('1.0'.split())
usage: PROG [-h] [-x {a,b,c} [{a,b,c} ...]] [-y {0,1,2}] {0.5,1.5}
PROG: error: argument z: invalid choice: 1.0 (choose from 0.5, 1.5)
```

Note that if choices is supplied for an argument that consumes multiple arg strings, each arg string will be checked against those choices.

2.1.9 Sharing arguments

The argparse module allows you to construct simple inheritance hierarchies of parsers when it's convenient to have multiple parsers that share some of the same arguments:

```
>>> foo_parser = argparse.ArgumentParser(add_help=False)
>>> foo_parser.add_argument('--foo')
>>> bar_parser = argparse.ArgumentParser(add_help=False)
>>> bar_parser.add_argument('bar')
>>> foo_bar_baz_parser = argparse.ArgumentParser(
...     parents=[foo_parser, bar_parser])
>>> foo_bar_baz_parser.add_argument('--baz')
>>> foo_bar_baz_parser.parse_args('--foo 1 XXX --baz 2'.split())
Namespace(bar='XXX', baz='2', foo='1')
```

If you end up with a lot of parsers (as may happen if you make extensive use of subparsers), the `parents` argument can help dramatically reduce the code duplication.

2.1.10 Suppress anything

Both default values and help strings can be suppressed in argparse. Simply provide `argparse.SUPPRESS` to the appropriate keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--secret', help=argparse.SUPPRESS)
>>> parser.add_argument('-d', default=argparse.SUPPRESS)
```

Note that when help for an argument is suppressed, that option will not be displayed in usage or help messages:

```
>>> parser.print_help()
usage: PROG [-h] [-d D]

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

And when a default is suppressed, the object returned by `parse_args()` will only include an attribute for the argument if the argument was actually present in the arg strings:

```
>>> parser.parse_args('--secret value'.split())
Namespace(secret='value')
>>> parser.parse_args('-d value'.split())
Namespace(d='value', secret=None)
```

2.2 Upgrading optparse code

Originally, the argparse module had attempted to maintain compatibility with optparse. However, optparse was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messges. When most everything in optparse had either been copy-pasted over or monkey-patched, it no longer seemed worthwhile to try to maintain the backwards compatibility.

A partial upgrade path from optparse to argparse:

- Replace all `add_option()` calls with `add_argument()` calls.
- Replace `options, args = parser.parse_args()` with `args = parser.parse_args()` and add additional `add_argument()` calls for the positional arguments.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace `Values` with `Namespace` and `OptionError/OptionValueError` with `ArgumentError`.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard python syntax to use dictionaries to format strings, that is to say, `%(default)s` and `%(prog)s`.

CHAPTER

THREE

API DOCUMENTATION

3.1 ArgumentParser objects

class **`ArgumentParser`**(*[description], [epilog], [prog], [usage], [version], [add_help], [argument_default], [parents], [prefix_chars], [conflict_handler], [formatter_class]*)

Create a new :class:ArgumentParser object. Each parameter has its own more detailed description below, but in short they are:

- description - Text to display before the argument help.
- epilog - Text to display after the argument help.
- version - A version number used to add a -v/–version option to the parser.
- add_help - Add a -h/–help option to the parser. (default: True)
- argument_default - Set the global default value for arguments. (default: None)
- parents - A list of :class:ArgumentParser objects whose arguments should also be included.
- prefix_chars - The set of characters that prefix optional arguments. (default: ‘-‘)
- fromfile_prefix_chars - The set of characters that prefix files from which additional arguments should be read. (default: None)
- formatter_class - A class for customizing the help output.
- conflict_handler - Usually unnecessary, defines strategy for resolving conflicting optionals.
- prog - Usually unnecessary, the name of the program (default: `sys.argv[0]`)
- usage - Usually unnecessary, the string describing the program usage (default: generated)

The following sections describe how each of these are used.

3.1.1 description

Most calls to the ArgumentParser constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

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

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the formatter_class argument.

3.1.2 epilog

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the `epilog=` argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

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

And that's how you'd foo a bar
```

As with the description argument, the `epilog=` text is by default line-wrapped, but this behavior can be adjusted with the formatter_class argument to ArgumentParser.

3.1.3 version

Programs which want to display the program version at the command line can supply a version message as the `version=` argument to ArgumentParser. This will add a `-v/--version` option to the ArgumentParser that can be invoked to print the version string:

```
>>> parser = argparse.ArgumentParser(prog='PROG', version='%(prog)s 3.5')
>>> parser.print_help()
usage: PROG [-h] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  show program's version number and exit
>>> parser.parse_args(['-v'])
PROG 3.5
```

Note you can use the `%(prog)s` format specifier to insert the program name into the version string.

3.1.4 add_help

By default, ArgumentParser objects add a `-h/--help` option which simply displays the parser's help message. For example, consider a file named `myprogram.py` containing the following code:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

If `-h` or `--help` is supplied is at the command-line, the ArgumentParser help will be printed:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

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

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the `add_help=` argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
  --foo FOO  foo help
```

3.1.5 prefix_chars

Most command-line options will use `'-'` as the prefix, e.g. `-f/--foo`. Parsers that need to support additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the ArgumentParser constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The `prefix_chars=` argument defaults to `'-'`. Supplying a set of characters that does not include `'-'` will cause `-f/--foo` options to be disallowed. Note that most parent parsers will specify `add_help() =False`. Otherwise, the ArgumentParser will see two `-h/--help` options (one in the parent and one in the child) and raise an error.

3.1.6 fromfile_prefix_chars

Sometimes, e.g. for particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the ArgumentParser constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> open('args.txt', 'w').write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must be one per line (with each whole line being considered a single argument) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

The fromfile_prefix_chars= argument defaults to None, meaning that arguments will never be treated as file references.

3.1.7 argument_default

Generally, argument defaults are specified either by passing a default to add_argument() or by calling the set_defaults() methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the argument_default= keyword argument to ArgumentParser. For example, to globally suppress attribute creation on parse_args() calls, we supply argument_default=SUPPRESS:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

3.1.8 parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, you can define a single parser with all the shared arguments and then use the parents= argument to ArgumentParser to have these "inherited". The parents= argument takes a list of ArgumentParser objects, collects all the positional and optional actions from them, and adds these actions to the ArgumentParser object being constructed:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

3.1.9 formatter_class

ArgumentParser objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are three such classes: argparse.RawDescriptionHelpFormatter, argparse.RawTextHelpFormatter and argparse.ArgumentDefaultsHelpFormatter. The first two allow more control over how textual descriptions are displayed, while the last automatically adds information about argument default values.

By default, ArgumentParser objects line-wrap the description and epilog texts in command-line help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
```

```
...         was indented weird
...             but that is okay'''),
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

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

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

When you have description and epilog that is already correctly formatted and should not be line-wrapped, you can indicate this by passing `argparse.RawDescriptionHelpFormatter` as the `formatter_class=` argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
    I have indented it
    exactly the way
    I want it

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

If you want to maintain whitespace for all sorts of help text (including argument descriptions), you can use `argparse.RawTextHelpFormatter`.

The other formatter class available, `argparse.ArgumentDefaultsHelpFormatter`, will add information about the default value of each of the arguments:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]
```

```
positional arguments:
  bar         BAR! (default: [1, 2, 3])

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   FOO! (default: 42)
```

3.1.10 conflict_handler

ArgumentParser objects do not allow two actions with the same option string. By default, ArgumentParser objects will raise an exception if you try to create an argument with an option string that is already in use:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
  ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Sometimes (e.g. when using parents) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value `'resolve'` can be supplied to the `conflict_handler=` argument of ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  -f FOO      old foo help
  --foo FOO   new foo help
```

Note that ArgumentParser objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/--foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

3.1.11 prog

By default, ArgumentParser objects use `sys.argv[0]` to determine how to display the name of the program in help messages. This default is almost always what you want because it will make the help messages match what your users have typed at the command line. For example, consider a file named `myprogram.py` with the following code:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

The help for this program will display `myprogram.py` as the program name (regardless of where the program was invoked from):

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo help
$ cd ..
$ python subdir\myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

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

To change this default behavior, another value can be supplied using the `prog=` argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

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

Note that the program name, whether determined from `sys.argv[0]` or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  --foo FOO   foo of the myprogram program
```

3.1.12 usage

By default, ArgumentParser objects calculate the usage message from the arguments it contains:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
  bar          bar help

optional arguments:
  -h, --help   show this help message and exit
  --foo [FOO]  foo help
```

If the default usage message is not appropriate for your application, you can supply your own usage message using the `usage=` keyword argument to ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
  bar           bar help

optional arguments:
  -h, --help    show this help message and exit
  --foo [FOO]   foo help
```

Note you can use the `%(prog)s` format specifier to fill in the program name in your usage messages.

3.2 The add_argument() method

add_argument(*name or flags..., [action], [nargs], [const], [default], [type], [choices], [required], [help], [metavar], [dest]*)

Define how a single command line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- •name or flags - Either a name or a list of option strings, e.g. `foo` or `-f, --foo`
- •action - The basic type of action to be taken when this argument is encountered at the command-line.
- •nargs - The number of command-line arguments that should be consumed.
- •const - A constant value required by some action and nargs selections.
- •default - The value produced if the argument is absent from the command-line.
- •type - The type to which the command-line arg should be converted.
- •choices - A container of the allowable values for the argument.
- •required - Whether or not the command-line option may be omitted (optionals only).
- •help - A brief description of what the argument does.
- •metavar - A name for the argument in usage messages.
- •dest - The name of the attribute to be added to the object returned by `parse_args()`.

The following sections describe how each of these are used.

3.2.1 name or flags

The `add_argument()` method needs to know whether you're expecting an optional argument, e.g. `-f` or `--foo`, or a positional argument, e.g. a list of filenames. The first arguments passed to `add_argument()` must therefore be either a series of flags, or a simple argument name. For example, an optional argument could be created like:

```
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```
>>> parser.add_argument('bar')
```

When `parse_args()` is called, optional arguments will be identified by the - prefix, and the remaining arguments will be assumed to be positional:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: too few arguments
```

3.2.2 action

`ArgumentParser` objects associate command-line args with actions. These actions can do just about anything with the command-line args associated with them, though most actions simply add an attribute to the object returned by `parse_args()`. When you specify a new argument using the `add_argument()` method, you can indicate how the command-line args should be handled by specifying the `action` keyword argument. The supported actions are:

- `'store'` - This just stores the argument's value. This is the default action. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- `'store_const'` - This stores the value specified by the const keyword argument. Note that the const keyword argument defaults to `None`, so you'll almost always need to provide a value for it. The `'store_const'` action is most commonly used with optional arguments that specify some sort of flag. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args('--foo'.split())
  Namespace(foo=42)
  ```

- `'store_true'` and `'store_false'` - These store the values `True` and `False` respectively. These are basically special cases of `'store_const'`. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_true')
  >>> parser.add_argument('--bar', action='store_false')
  >>> parser.parse_args('--foo --bar'.split())
  Namespace(bar=False, foo=True)
  ```

- `'append'` - This stores a list, and appends each argument value to the list. This is useful when you want to allow an option to be specified multiple times. Example usage:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='append')
  >>> parser.parse_args('--foo 1 --foo 2'.split())
  Namespace(foo=['1', '2'])
  ```

- `'append_const'` - This stores a list, and appends the value specified by the const keyword argument to the list. Note that the const keyword argument defaults to `None`, so you'll almost always need to provide a value

for it. The 'append_const' action is typically useful when you want multiple arguments to store constants to the same list, for example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<type 'str'>, <type 'int'>])
```

You can also specify an arbitrary action by passing an object that implements the Action API. The easiest way to do this is to extend argparse.Action, supplying an appropriate __call__ method. The __call__ method accepts four parameters:

- parser - The ArgumentParser object which contains this action.
- namespace - The namespace object that will be returned by parse_args(). Most actions add an attribute to this object.
- values - The associated command-line args, with any type-conversions applied. (Type-conversions are specified with the type keyword argument to add_argument().
- option_string - The option string that was used to invoke this action. The option_string argument is optional, and will be absent if the action is associated with a positional argument.

So for example:

```
>>> class FooAction(argparse.Action):
...     def __call__(self, parser, namespace, values, option_string=None):
...     print '%r %r %r' % (namespace, values, option_string)
...     setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

3.2.3 nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. In the situations where you'd like to associate a different number of command-line arguments with a single action, you can use the nargs keyword argument to add_argument(). The supported values are:

- N (an integer). N args from the command-line will be gathered together into a list. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

Note that `nargs=1` produces a list of one item. This is different from the default, in which the item is produced by itself.

- `'?'`. One arg will be consumed from the command-line if possible, and produced as a single item. If no command-line arg is present, the value from default will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line arg. In this case the value from const will be produced. Some examples to illustrate this:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args('XX --foo YY'.split())
Namespace(bar='XX', foo='YY')
>>> parser.parse_args('XX --foo'.split())
Namespace(bar='XX', foo='c')
>>> parser.parse_args(''.split())
Namespace(bar='d', foo='d')
```

One of the more common uses of `nargs='?'` is to allow optional input and output files:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'), default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'), default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<open file 'input.txt', mode 'r' at 0x...>, outfile=<open file 'output.txt', mo
>>> parser.parse_args([])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>, outfile=<open file '<stdout>', mode '
```

- `'*'`. All command-line args present are gathered into a list. Note that it generally doesn't make much sense to have more than one positional argument with `nargs='*'`, but multiple optional arguments with `nargs='*'` is possible. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`. Just like `'*'`, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn't at least one command-line arg present. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args('a b'.split())
Namespace(foo=['a', 'b'])
>>> parser.parse_args(''.split())
usage: PROG [-h] foo [foo ...]
PROG: error: too few arguments
```

If the `nargs` keyword argument is not provided, the number of args consumed is determined by the action. Generally this means a single command-line arg will be consumed and a single item (not a list) will be produced.

3.2.4 const

The `const` argument of `add_argument()` is used to hold constant values that are not read from the command line but are required for the various ArgumentParser actions. The two most common uses of it are:

- When `add_argument()` is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by `parse_args()`. See the action description for examples.
- When `add_argument()` is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line args. When parsing the command-line, if the option string is encountered with no command-line arg following it, the value of `const` will be assumed instead. See the nargs description for examples.

The `const` keyword argument defaults to `None`.

3.2.5 default

All optional arguments and some positional arguments may be omitted at the command-line. The `default` keyword argument of `add_argument()`, whose value defaults to `None`, specifies what value should be used if the command-line arg is not present. For optional arguments, the `default` value is used when the option string was not present at the command line:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args('--foo 2'.split())
Namespace(foo='2')
>>> parser.parse_args(''.split())
Namespace(foo=42)
```

For positional arguments with nargs `='?'` or `'*'`, the `default` value is used when no command-line arg was present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args('a'.split())
Namespace(foo='a')
>>> parser.parse_args(''.split())
Namespace(foo=42)
```

If you don't want to see an attribute when an option was not present at the command line, you can supply `default=argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

3.2.6 type

By default, ArgumentParser objects read command-line args in as simple strings. However, quite often the command-line string should instead be interpreted as another type, e.g. `float`, `int` or `file`. The `type` keyword argument

of `add_argument()` allows any necessary type-checking and type-conversions to be performed. Many common builtin types can be used directly as the value of the `type` argument:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=file)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<open file 'temp.txt', mode 'r' at 0x...>, foo=2)
```

To ease the use of various types of files, the argparse module provides the factory FileType which takes the `mode=` and `bufsize=` arguments of the `file` object. For example, `FileType('w')` can be used to create a writable file:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```

If you need to do some special type-checking or type-conversions, you can provide your own types by passing to `type=` a callable that takes a single string argument and returns the type-converted value:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...     raise TypeError()
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args('9'.split())
Namespace(foo=9)
>>> parser.parse_args('7'.split())
usage: PROG [-h] foo
PROG: error: argument foo: invalid perfect_square value: '7'
```

Note that if your type-checking function is just checking for a particular set of values, it may be more convenient to use the choices keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=xrange(5, 10))
>>> parser.parse_args('7'.split())
Namespace(foo=7)
>>> parser.parse_args('11'.split())
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

See the choices section for more details.

3.2.7 choices

Some command-line args should be selected from a restricted set of values. ArgumentParser objects can be told about such sets of values by passing a container object as the `choices` keyword argument to `add_argument()`. When the command-line is parsed with `parse_args()`, arg values will be checked, and an error message will be displayed if the arg was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', choices='abc')
>>> parser.parse_args('c'.split())
Namespace(foo='c')
>>> parser.parse_args('X'.split())
usage: PROG [-h] {a,b,c}
PROG: error: argument foo: invalid choice: 'X' (choose from 'a', 'b', 'c')
```

Note that inclusion in the `choices` container is checked after any type conversions have been performed, so the type of the objects in the `choices` container should match the type specified:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=complex, choices=[1, 1j])
>>> parser.parse_args('1j'.split())
Namespace(foo=1j)
>>> parser.parse_args('-- -4'.split())
usage: PROG [-h] {1,1j}
PROG: error: argument foo: invalid choice: (-4+0j) (choose from 1, 1j)
```

Any object that supports the `in` operator can be passed as the `choices` value, so `dict` objects, `set` objects, custom containers, etc. are all supported.

3.2.8 required

In general, the argparse module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command-line. To change this behavior, i.e. to make an option *required*, the value `True` should be specified for the `required=` keyword argument to `add_argument()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

As the example shows, if an option is marked as `required`, `parse_args()` will report an error if that option is not present at the command line.

Warning: Required options are generally considered bad form - normal users expect *options* to be *optional*. You should avoid the use of required options whenever possible.

3.2.9 help

A great command-line interface isn't worth anything if your users can't figure out which option does what. So for the end-users, `help` is probably the most important argument to include in your `add_argument()` calls. The `help` value should be a string containing a brief description of what the argument specifies. When a user requests help (usually by using `-h` or `--help` at the command-line), these `help` descriptions will be displayed with each argument:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...         help='foo the bars before frobbling')
```

```
>>> parser.add_argument('bar', nargs='+',
...         help='one of the bars to be frobbled')
>>> parser.parse_args('-h'.split())
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
  bar     one of the bars to be frobbled

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo the bars before frobbling
```

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument default. The available specifiers include the program name, `%(prog)s` and most keyword arguments to `add_argument()`, e.g. `%(default)s`, `%(type)s`, etc.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...         help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
  bar     the bar to frobble (default: 42)

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

3.2.10 metavar

When ArgumentParser objects generate help messages, they need some way to refer to each expected argument. By default, ArgumentParser objects use the dest value as the "name" of each object. By default, for positional argument actions, the dest value is used directly, and for optional argument actions, the dest value is uppercased. So if we have a single positional argument with `dest='bar'`, that argument will be referred to as `bar`. And if we have a single optional argument `--foo` that should be followed by a single command-line arg, that arg will be referred to as `FOO`. You can see this behavior in the example below:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
  bar

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

If you would like to provide a different name for your argument in help messages, you can supply a value for the `metavar` keyword argument to `add_argument()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
  XXX

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

Note that `metavar` only changes the *displayed* name - the name of the attribute on the `parse_args()` object is still determined by the dest value.

Different values of `nargs` may cause the metavar to be used multiple times. If you'd like to specify a different display name for each of the arguments, you can provide a tuple to `metavar`:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
  -h, --help     show this help message and exit
  -x X X
  --foo bar baz
```

3.2.11 dest

Most ArgumentParser actions add some value as an attribute of the object returned by `parse_args()`. The name of this attribute is determined by the `dest` keyword argument of `add_argument()`. For positional argument actions, `dest` is normally supplied as the first argument to `add_argument()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args('XXX'.split())
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. ArgumentParser objects generate the value of `dest` by taking the first long option string and stripping away the initial `'--'` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `'-'` character. Any internal `'-'` characters will be converted to `'_'` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
```

```
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

If you would like to use a different attribute name from the one automatically inferred by the ArgumentParser, you can supply it with an explicit `dest` parameter:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

3.3 The parse_args() method

parse_args(*[args], [namespace]*)

Convert the strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to `add_argument()` determine exactly what objects are created and how they are assigned. See the documentation for `add_argument()` for details.

By default, the arg strings are taken from `sys.argv`, and a new empty `Namespace` object is created for the attributes.

3.3.1 Option value syntax

The `parse_args()` method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args('-x X'.split())
Namespace(foo=None, x='X')
>>> parser.parse_args('--foo FOO'.split())
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), you may also pass the option and value as a single command line argument, using = to separate them:

```
>>> parser.parse_args('--foo=FOO'.split())
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), you may simply concatenate the option and its value:

```
>>> parser.parse_args('-xX'.split())
Namespace(foo=None, x='X')
```

You can also combine several short options together, using only a single - prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
```

```
>>> parser.parse_args('-xyzZ'.split())
Namespace(x=True, y=True, z='Z')
```

3.3.2 Invalid arguments

While parsing the command-line, `parse_args` checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

3.3.3 Arguments containing `"-"`

The `parse_args` method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line arg `'-1'` could either be an attempt to specify an option or an attempt to provide a positional argument. The `parse_args` method is cautious here: positional arguments may only begin with `'-'` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
```

```
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with `'-'` and don't look like negative numbers, you can insert the pseudo-argument `'--'` which tells `parse_args` that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

3.3.4 Argument abbreviations

The `parse_args()` method allows you to abbreviate long options if the abbreviation is unambiguous:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

As you can see above, you will get an error if you pick a prefix that could refer to more than one option.

3.3.5 Beyond `sys.argv`

Sometimes it may be useful to have an ArgumentParser parse args other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args`. You may have noticed that the examples in the argparse documentation have made heavy use of this calling style - it is much easier to use at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=xrange(10),
...  nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...   default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args('1 2 3 4 --sum'.split())
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

3.3.6 Custom namespaces

It may also be useful to have an ArgumentParser assign attributes to an already existing object, rather than the newly-created Namespace object that is normally used. This can be achieved by specifying the `namespace=` keyword argument:

```
>>> class C(object):
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

3.4 Other methods

3.4.1 Partial parsing

parse_known_args(*[args], [namespace]*)

Sometimes a script may only parse a few of the command line arguments, passing the remaining arguments on to another script or program. In these cases, the `parse_known_args()` method can be useful. It works much like `parse_args()` except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

3.4.2 Printing help

In most typical applications, `parse_args()` will take care of formatting and printing any usage or error messages. However, should you want to format or print these on your own, several methods are available:

print_usage([file]):()
: Print a brief description of how the `ArgumentParser` should be invoked on the command line. If `file` is not present, `sys.stderr` is assumed.

print_help([file]):()
: Print a help message, including the program usage and information about the arguments registered with the `ArgumentParser`. If `file` is not present, `sys.stderr` is assumed.

There are also variants of these methods that simply return a string instead of printing it:

format_usage():()
: Return a string containing a brief description of how the `ArgumentParser` should be invoked on the command line.

format_help():()
: Return a string containing a help message, including the program usage and information about the arguments

registered with the `ArgumentParser`.

3.4.3 Parser defaults

set_defaults(***kwargs*)

Most of the time, the attributes of the object returned by `parse_args()` will be fully determined by inspecting the command-line args and the argument actions described in your `add_argument()` calls. However, sometimes it may be useful to add some additional attributes that are determined without any inspection of the command-line. The `set_defaults()` method allows you to do this:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults. So if you set a parser-level default for a name that matches an argument, the old argument default will no longer be used:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when you're working with multiple parsers. See the `add_subparsers()` method for an example of this type.

3.4.4 Sub-commands

add_subparsers()

A lot of programs split up their functionality into a number of sub-commands, for example, the `svn` program can invoke sub-commands like `svn checkout`, `svn update`, `svn commit`, etc. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. ArgumentParser objects support the creation of such sub-commands with the `add_subparsers()` method. The `add_subparsers()` method is normally called with no arguments and returns an special action object. This object has a single method, `add_parser`, which takes a command name and any ArgumentParser constructor arguments, and returns an ArgumentParser object that can be modified as usual.

Some example usage:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
```

```
>>> # parse some arg lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by `parse_args()` will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `"a"` command is specified, only the `foo` and `bar` attributes are present, and when the `"b"` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (You can however supply a help message for each subparser command by suppling the `help=` argument to `add_parser` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
a     a help
b     b help

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

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

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

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The `add_subparsers()` method also supports `title` and `description` keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

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

```
subcommands:
  valid subcommands

  {foo,bar}   additional help
```

One particularly effective way of handling sub-commands is to combine the use of the `add_subparsers()` method with calls to `set_defaults()` so that each subparser knows which Python function it should execute. For example:

```
>>> # sub-command functions
>>> def foo(args):
...     print args.x * args.y
...
>>> def bar(args):
...     print '((%s))' % args.z
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let `parse_args()` do all the work for you, and then just call the appropriate function after the argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if you find it necessary to check the name of the subparser that was invoked, you can always provide a `dest` keyword argument to the `add_subparsers()` call:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

3.4.5 Argument groups

add_argument_group(*[title], [description]*)

By default, ArgumentParser objects group command-line arguments into "positional arguments" and "optional arguments" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The `add_argument_group()` method returns an argument group object which has an `add_argument()` method just like a regular ArgumentParser objects. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts `title` and `description` arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

Note that any arguments not in your user defined groups will end up back in the usual "positional arguments" and "optional arguments" sections.

3.4.6 Mutual exclusion

add_mutually_exclusive_group(*[required=False]*)

Sometimes, you need to make sure that only one of a couple different options is specified on the command line. You can create groups of such mutually exclusive arguments using the `add_mutually_exclusive_group()` method. When `parse_args()` is called, argparse will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
```

```
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The `add_mutually_exclusive_group()` method also accepts a `required` argument, to indicate that at least one of the mutually exclusive arguments is required:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the `title` and `description` arguments of `add_argument_group()`. This may change in the future however, so you are *strongly* recommended to specify `required` as a keyword argument if you use it.

3.5 Other utilities

3.5.1 FileType objects

class **`FileType`**(*mode='r', bufsize=None*)

The `FileType` factory creates objects that can be passed to the type argument of `add_argument()`. Arguments that have `FileType` objects as their type will open command-line args as files with the requested modes and buffer sizes:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--output', type=argparse.FileType('wb', 0))
>>> parser.parse_args(['--output', 'out'])
Namespace(output=<open file 'out', mode 'wb' at 0x...>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into `sys.stdin` for readable `FileType` objects and `sys.stdout` for writable `FileType` objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<open file '<stdin>', mode 'r' at 0x...>)
```

CHAPTER

FOUR

EXAMPLE USAGE

The following simple example uses the argparse module to generate the command-line interface for a Python program that sums its command-line arguments and writes them to a log file:

```
import argparse
import sys

if __name__ == '__main__':

    # create the parser
    parser = argparse.ArgumentParser(
        description='Sum the integers on the command line.')

    # add the arguments
    parser.add_argument(
        'integers', metavar='int', type=int, nargs='+',
        help='one of the integers to be summed')
    parser.add_argument(
        '--log', type=argparse.FileType('w'), default=sys.stdout,
        help='the file where the sum should be written '
             '(default: write the sum to stdout)')

    # parse the command line
    args = parser.parse_args()

    # write out the sum
    args.log.write('%s\n' % sum(args.integers))
    args.log.close()
```

Assuming the Python code above is saved into a file called `scriptname.py`, it can be run at the command line and provides useful help messages:

```
$ scriptname.py -h
usage: scriptname.py [-h] [--log LOG] int [int ...]

Sum the integers on the command line.

positional arguments:
  int         one of the integers to be summed

optional arguments:
  -h, --help  show this help message and exit
  --log LOG   the file where the sum should be written (default: write the sum
              to stdout)
```

When run with the appropriate arguments, it writes the sum of the command-line integers to the specified log file:

```
$ scriptname.py --log=log.txt 1 1 2 3 5 8
$ more log.txt
20
```