:mod:`argparse` --- Parser for command-line options, arguments and sub-commands

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 4)

Unknown directive type "module".

```
.. module:: argparse
   :synopsis: Command-line option and argument parsing library.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 7)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Steven Bethard <steven.bethard@gmail.com>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 8)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Steven Bethard <steven.bethard@gmail.com>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 10)

Unknown directive type "versionadded".

```
.. versionadded:: 3.2
```

Source code: :source:`Lib/argparse.py`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 12); *backlink*

Unknown interpreted text role "source".

The :mod:`argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and :mod:`argparse` will figure out how to parse those out of :data:`sys.argv`. The :mod:`argparse` module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 22); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 22); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 22); *backlink*

Unknown interpreted text role "data".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 22); *backlink*

Unknown interpreted text role "mod".

Tutorial

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the :ref`argparse tutorial <argparse-tutorial>`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 18); *backlink*

Unknown interpreted text role "ref".

Example

The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
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 the Python code above is saved into a file called `prog.py`, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]
```

```
Process some integers.

positional arguments:
 N           an integer for the accumulator

options:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

The following sections walk you through this example.

Creating a parser

The first step in using the :mod:`argparse` is creating an :class:`ArgumentParser` object:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 89); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 89); *backlink*

Unknown interpreted text role "class".

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

The :class:`ArgumentParser` object will hold all the information necessary to parse the command line into Python data types.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 94); *backlink*

Unknown interpreted text role "class".

Adding arguments

Filling an :class:`ArgumentParser` with information about program arguments is done by making calls to the :meth:`~ArgumentParser.add_argument` method. Generally, these calls tell the :class:`ArgumentParser` how to take the strings on the command line and turn them into objects. This information is stored and used when :meth:`~ArgumentParser.parse_args` is called. For example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 101); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 101); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 101); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 101); *backlink*

Unknown interpreted text role "meth".

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

Later, calling :meth:`~ArgumentParser.parse_args` will return an object with two attributes, `integers` and `accumulate`. The `integers` attribute will be a list of one or more ints, and the `accumulate` attribute will be either the :func:`sum` function, if `--sum` was specified at the command line, or the :func:`max` function if it was not.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 113); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 113); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst, line 113); *backlink*

Unknown interpreted text role "func".

Parsing arguments

:class:`ArgumentParser` parses arguments through the :meth:`~ArgumentParser.parse_args` method. This will inspect the command

line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple :class:`Namespace` object will be built up from attributes parsed out of the command line:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 123); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 123); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 123); *backlink*

Unknown interpreted text role "class".

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, :meth:`~ArgumentParser.parse_args` will typically be called with no arguments, and the :class:`ArgumentParser` will automatically determine the command-line arguments from :data:`sys.argv`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 132); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 132); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 132); *backlink*

Unknown interpreted text role "data".

ArgumentParser objects

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 140)

Invalid class attribute value for "class" directive: "ArgumentParser(prog=None, usage=None, description=None, \ epilog=None, parents=[], \ formatter_class=argparse.HelpFormatter, \ prefix_chars='-', fromfile_prefix_chars=None, \ argument_default=None, conflict_handler='error', \ add_help=True, allow_abbrev=True, exit_on_error=True)".

```
.. class:: ArgumentParser(prog=None, usage=None, description=None, \
                          epilog=None, parents=[], \
                          formatter_class=argparse.HelpFormatter, \
                          prefix_chars='-', fromfile_prefix_chars=None, \
                          argument_default=None, conflict_handler='error', \
                          add_help=True, allow_abbrev=True, exit_on_error=True)

   Create a new :class:`ArgumentParser` object. All parameters should be passed
   as keyword arguments. Each parameter has its own more detailed description
   below, but in short they are:

   * prog_ - The name of the program (default:
     ``os.path.basename(sys.argv[0])``)

   * usage_ - The string describing the program usage (default: generated from
     arguments added to parser)

   * description_ - Text to display before the argument help (default: none)

   * epilog_ - Text to display after the argument help (default: none)

   * parents_ - A list of :class:`ArgumentParser` objects whose arguments should
     also be included

   * formatter_class_ - A class for customizing the help output

   * 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``)

   * argument_default_ - The global default value for arguments
     (default: ``None``)

   * conflict_handler_ - The strategy for resolving conflicting optionals
     (usually unnecessary)

   * add_help_ - Add a ``-h/--help`` option to the parser (default: ``True``)

   * allow_abbrev_ - Allows long options to be abbreviated if the
     abbreviation is unambiguous. (default: ``True``)

   * exit_on_error_ - Determines whether or not ArgumentParser exits with
     error info when an error occurs. (default: ``True``)

   .. versionchanged:: 3.5
      *allow_abbrev* parameter was added.

   .. versionchanged:: 3.8
      In previous versions, *allow_abbrev* also disabled grouping of short
      flags such as ``-vv`` to mean ``-v -v``.

   .. versionchanged:: 3.9
      *exit_on_error* parameter was added.
```

The following sections describe how each of these are used.

prog

By default, :class:`ArgumentParser` objects use `sys.argv[0]` to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named `myprogram.py` with the following code:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 202); *backlink***
>
> Unknown interpreted text role "class".

```
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]

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

options:
 -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 :class:`ArgumentParser`:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 232); *backlink***
>
> Unknown interpreted text role "class".

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

options:
 -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]

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

usage

By default, :class:`ArgumentParser` calculates the usage message from the arguments it contains:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 261); *backlink***
>
> Unknown interpreted text role "class".

```
>>> 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

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

The default message can be overridden with the `usage=` keyword argument:

```
>>> 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

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

The `%(prog)s` format specifier is available to fill in the program name in your usage messages.

description

Most calls to the :class:`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:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 299); *backlink***
>
> Unknown interpreted text role "class".

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

A foo that bars
```

```
options:
 -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.

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 :class:`ArgumentParser`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 321); backlink

Unknown interpreted text role "class".

```
>>> 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

options:
 -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 :class:`ArgumentParser`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 338); backlink

Unknown interpreted text role "class".

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to `parents=` argument to :class:`ArgumentParser` can be used. The `parents=` argument takes a list of :class:`ArgumentParser` objects, collects all the positional and optional actions from them, and adds these actions to the :class:`ArgumentParser` object being constructed:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 346); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 346); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 346); backlink

Unknown interpreted text role "class".

```
>>> 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)
```

Note that most parent parsers will specify `add_help=False`. Otherwise, the :class:`ArgumentParser` will see two `-h/--help` options (one in the parent and one in the child) and raise an error.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 366); backlink

Unknown interpreted text role "class".

Note

You must fully initialize the parsers before passing them via `parents=`. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

formatter_class

:class:`ArgumentParser` objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 379); backlink

Unknown interpreted text role "class".

:class:`RawDescriptionHelpFormatter` and :class:`RawTextHelpFormatter` give more control over how textual descriptions are displayed. By default, :class:`ArgumentParser` objects line-wrap the description and epilog texts in command-line help messages:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 388); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 388); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 388); *backlink*

Unknown interpreted text role "class".

```
>>> 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

options:
 -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
```

Passing :class:`RawDescriptionHelpFormatter` as `formatter_class=` indicates that description and epilog are already correctly formatted and should not be line-wrapped:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 413); *backlink*

Unknown interpreted text role "class".

```
>>> 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

options:
 -h, --help  show this help message and exit
```

:class:`RawTextHelpFormatter` maintains whitespace for all sorts of help text, including argument descriptions. However, multiple new lines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 439); *backlink*

Unknown interpreted text role "class".

:class:`ArgumentDefaultsHelpFormatter` automatically adds information about default values to each of the argument help messages:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 444); *backlink*

Unknown interpreted text role "class".

```
>>> 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 ...]

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

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

:class:`MetavarTypeHelpFormatter` uses the name of the type argument for each argument as the display name for its values (rather than using the dest as the regular formatter does):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 462); *backlink*

Unknown interpreted text role "class".

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
```

```
    --foo int
```

prefix_chars

Most command-line options will use - as the prefix, e.g. `-f/--foo`. Parsers that need to support different or 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.

fromfile_prefix_chars

Sometimes, for example when dealing with a 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 :class:`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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 505); *backlink*

Unknown interpreted text role "class".

```
>>> with open('args.txt', 'w') as fp:
...     fp.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 by default be one per line (but see also :meth:`~ArgumentParser.convert_arg_line_to_args`) 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']`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 519); *backlink*

Unknown interpreted text role "meth".

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

argument_default

Generally, argument defaults are specified either by passing a default to :meth:`~ArgumentParser.add_argument` or by calling the :meth:`~ArgumentParser.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 :class:`ArgumentParser`. For example, to globally suppress attribute creation on :meth:`~ArgumentParser.parse_args` calls, we supply `argument_default=SUPPRESS`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 532); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 532); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 532); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 532); *backlink*

Unknown interpreted text role "meth".

```
>>> 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()
```

allow_abbrev

Normally, when you pass an argument list to the :meth:`~ArgumentParser.parse_args` method of an :class:`ArgumentParser`, it :ref:`recognizes abbreviations <prefix-matching>` of long options.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 554); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 554); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 554); *backlink*

Unknown interpreted text role "ref".

This feature can be disabled by setting `allow_abbrev` to `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 567)

Unknown directive type "versionadded".

```
.. versionadded:: 3.5
```

conflict_handler

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 573); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 573); *backlink*

Unknown interpreted text role "class".

```
>>> 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 :class:`ArgumentParser`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 585); *backlink*

Unknown interpreted text role "class".

```
>>> 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]

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

Note that :class:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 601); *backlink*

Unknown interpreted text role "class".

add_help

By default, ArgumentParser objects add an 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 at the command line, the ArgumentParser help will be printed:

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

options:
 -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 :class:`ArgumentParser`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 631); *backlink*

Unknown interpreted text role "class".

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

options:
 --foo FOO  foo help
```

The help option is typically `-h/--help`. The exception to this is if the `prefix_chars=` is specified and does not include `-`, in which case `-h` and `--help` are not valid options. In this case, the first character in `prefix_chars` is used to prefix the help options:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

options:
  +h, ++help  show this help message and exit
```

exit_on_error

Normally, when you pass an invalid argument list to the :meth:`~ArgumentParser.parse_args` method of an :class:`ArgumentParser`, it will exit with error info.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 660); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 660); *backlink*

Unknown interpreted text role "class".

If the user would like to catch errors manually, the feature can be enabled by setting `exit_on_error` to `False`:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None, const=None, default=None, type=<class 'int'>, choices=Non
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 676)

Unknown directive type "versionadded".

```
.. versionadded:: 3.9
```

The add_argument() method

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 682)

Unknown directive type "method".

```
.. method:: ArgumentParser.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 and if it is absent from the namespace object.

   * type_ - The type to which the command-line argument 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
     :meth:`parse_args`.
```

The following sections describe how each of these are used.

name or flags

The :meth:`~ArgumentParser.add_argument` method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to :meth:`~ArgumentParser.add_argument` must therefore be either a series of flags, or a simple argument name.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 722); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 722); *backlink*

Unknown interpreted text role "meth".

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 :meth:`~ArgumentParser.parse_args` is called, optional arguments will be identified by the - prefix, and the remaining arguments will be assumed to be positional:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 736); *backlink*

Unknown interpreted text role "meth".

```
>>> 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: the following arguments are required: bar
```

action

:class:`ArgumentParser` objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by :meth:`~ArgumentParser.parse_args`. The `action` keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 755); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 755); *backlink*

Unknown interpreted text role "meth".

- `'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`. 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'])
  Namespace(foo=42)
  ```

- `'store_true'` and `'store_false'` - These are special cases of `'store_const'` used for storing the values `True` and `False` respectively. In addition, they create default values of `False` and `True` respectively. For example:

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

- `'append'` - This stores a list, and appends each argument value to the list. This is useful 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`. The `'append_const'` action is typically useful when multiple arguments need 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=[<class 'str'>, <class 'int'>])
  ```

- `'count'` - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--verbose', '-v', action='count', default=0)
  >>> parser.parse_args(['-vvv'])
  Namespace(verbose=3)
  ```

  Note, the *default* will be `None` unless explicitly set to *0*.

- `'help'` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See :class:`ArgumentParser` for details of how the output is created.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 822); *backlink*

  Unknown interpreted text role "class".

- `'version'` - This expects a `version=` keyword argument in the :meth:`~ArgumentParser.add_argument` call, and prints version information and exits when invoked:

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 827); *backlink*

  Unknown interpreted text role "meth".

  ```
  >>> import argparse
  >>> parser = argparse.ArgumentParser(prog='PROG')
  >>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
  >>> parser.parse_args(['--version'])
  PROG 2.0
  ```

- `'extend'` - This stores a list, and extends each argument value to the list. Example usage:

  ```
  >>> parser = argparse.ArgumentParser()
  ```

```
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 846)
>
> Unknown directive type "versionadded".
>
> ```
> .. versionadded:: 3.8
> ```

You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The `BooleanOptionalAction` is available in `argparse` and adds support for boolean actions such as `--foo` and `--no-foo`:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 859)
>
> Unknown directive type "versionadded".
>
> ```
> .. versionadded:: 3.9
> ```

The recommended way to create a custom action is to extend :class:`Action`, overriding the `__call__` method and optionally the `__init__` and `format_usage` methods.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 861); *backlink*
>
> Unknown interpreted text role "class".

An example of a custom action:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     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')
```

For more details, see :class:`Action`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 885); *backlink*
>
> Unknown interpreted text role "class".

nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. The supported values are:

- `N` (an integer). `N` arguments 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.

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 907)
  >
  > Unknown directive type "index".
  >
  > ```
  > .. index:: single: ? (question mark); in argparse module
  > ```

- `'?'`. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument 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 argument. 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'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  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=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
  ```

```
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 941)

Unknown directive type "index".

```
.. index:: single: * (asterisk); in argparse module
```

- `'*'`. All command-line arguments 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'])
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 955)

Unknown directive type "index".

```
.. index:: single: + (plus); in argparse module
```

- `'+'`. 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 argument present. For example:

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

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

const

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 977); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 977); *backlink*

Unknown interpreted text role "class".

- When :meth:`~ArgumentParser.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 :meth:`~ArgumentParser.parse_args`. See the action description for examples. If `const` is not provided to :meth:`~ArgumentParser.add_argument`, it will receive a default value of `None`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 981); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 981); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 981); *backlink*

Unknown interpreted text role "meth".

- When :meth:`~ArgumentParser.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 arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of `const` will be assumed to be `None` instead. See the nargs description for examples.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 989); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 996)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.11
   ``const=None`` by default, including when ``action='append_const'`` or
   ``action='store_const'``.
```

default

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

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1003); *backlink*
>
> Unknown interpreted text role "meth".

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

If the target namespace already has an attribute set, the action *default* will not over write it:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any type conversion argument, if provided, before setting the attribute on the :class:`Namespace` return value. Otherwise, the parser uses the value as is:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1025); *backlink*
>
> Unknown interpreted text role "class".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with nargs equal to `?` or `*`, the `default` value is used when no command-line argument was present:

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

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:

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

type

By default, the parser reads command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, such as a :class:`float` or :class:`int`. The `type` keyword for :meth:`~ArgumentParser.add_argument` allows any necessary type-checking and type conversions to be performed.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1061); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1061); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1061); *backlink*
>
> Unknown interpreted text role "meth".

If the type keyword is used with the default keyword, the type converter is only applied if the default is a string.

The argument to `type` can be any callable that accepts a single string. If the function raises :exc:`ArgumentTypeError`, :exc:`TypeError`, or :exc:`ValueError`, the exception is caught and a nicely formatted error message is displayed. No other exception types are handled.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1070); *backlink*
>
> Unknown interpreted text role "exc".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1070); *backlink*
>
> Unknown interpreted text role "exc".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1070); *backlink*
>
> Unknown interpreted text role "exc".

Common built-in types and functions can be used as type converters:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1077)
>
> Unknown directive type "testcode".

```
.. testcode::

   import argparse
   import pathlib

   parser = argparse.ArgumentParser()
   parser.add_argument('count', type=int)
   parser.add_argument('distance', type=float)
   parser.add_argument('street', type=ascii)
   parser.add_argument('code_point', type=ord)
   parser.add_argument('source_file', type=open)
   parser.add_argument('dest_file', type=argparse.FileType('w', encoding='latin-1'))
   parser.add_argument('datapath', type=pathlib.Path)
```

User defined functions can be used as well:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1093)

Unknown directive type "doctest".

```
.. doctest::

   >>> def hyphenated(string):
   ...     return '-'.join([word[:4] for word in string.casefold().split()])
   ...
   >>> parser = argparse.ArgumentParser()
   >>> _ = parser.add_argument('short_title', type=hyphenated)
   >>> parser.parse_args(['"The Tale of Two Cities"'])
   Namespace(short_title='"the-tale-of-two-citi')
```

The :func:`bool` function is not recommended as a type converter. All it does is convert empty strings to `False` and non-empty strings to `True`. This is usually not what is desired.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1103); *backlink*

Unknown interpreted text role "func".

In general, the `type` keyword is a convenience that should only be used for simple conversions that can only raise one of the three supported exceptions. Anything with more interesting error-handling or resource management should be done downstream after the arguments are parsed.

For example, JSON or YAML conversions have complex error cases that require better reporting than can be given by the `type` keyword. A :exc:`~json.JSONDecodeError` would not be well formatted and a :exc:`FileNotFound` exception would not be handled at all.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1112); *backlink*

Unknown interpreted text role "exc".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1112); *backlink*

Unknown interpreted text role "exc".

Even :class:`~argparse.FileType` has its limitations for use with the `type` keyword. If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. In this case, it would be better to wait until after the parser has run and then use the :keyword:`with`-statement to manage the files.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1117); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1117); *backlink*

Unknown interpreted text role "keyword".

For type checkers that simply check against a fixed set of values, consider using the choices keyword instead.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the *choices* keyword argument to :meth:`~ArgumentParser.add_argument`. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1130); *backlink*

Unknown interpreted text role "meth".

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

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='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any container can be passed as the *choices* value, so :class:`list` objects, :class:`set` objects, and custom containers are all

supported.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1157); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1157); *backlink***
>
> Unknown interpreted text role "class".

Use of :class:`enum.Enum` is not recommended because it is difficult to control its appearance in usage, help, and error messages.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1160); *backlink***
>
> Unknown interpreted text role "class".

Formatted choices overrides the default *metavar* which is normally derived from *dest*. This is usually what you want because the user never sees the *dest* parameter. If this display isn't desirable (perhaps because there are many choices), just specify an explicit metavar.

required

In general, the :mod:`argparse` module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to :meth:`~ArgumentParser.add_argument`:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1172); *backlink***
>
> Unknown interpreted text role "mod".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1172); *backlink***
>
> Unknown interpreted text role "meth".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

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

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1185); *backlink***
>
> Unknown interpreted text role "meth".

> **Note**
>
> Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

help

The `help` value is a string containing a brief description of the argument. 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'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

options:
 -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 :meth:`~ArgumentParser.add_argument`, e.g. `%(default)s`, `%(type)s`, etc.:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1218); *backlink***
>
> Unknown interpreted text role "meth".

```
>>> 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)

options:
 -h, --help  show this help message and exit
```

As the help string supports %-formatting, if you want a literal `%` to appear in the help string, you must escape it as `%%`.

:mod:`argparse` supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS`:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1238); backlink

Unknown interpreted text role "mod".

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

options:
  -h, --help  show this help message and exit
```

metavar

When :class:`ArgumentParser` generates help messages, it needs 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, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1253); backlink

Unknown interpreted text role "class".

```
>>> 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

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

An alternative name can be specified with `metavar`:

```
>>> 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

options:
 -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 :meth:`~ArgumentParser.parse_args` object is still determined by the dest value.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1294); backlink

Unknown interpreted text role "meth".

Different values of `nargs` may cause the metavar to be used multiple times. Providing a tuple to `metavar` specifies a different display for each of the arguments:

```
>>> 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]

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

dest

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1317); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1317); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1317); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1317); backlink

Unknown interpreted text role "meth".

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

For optional argument actions, the value of `dest` is normally inferred from the option strings. :class:`ArgumentParser` generates 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:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1329); *backlink***
>
> Unknown interpreted text role "class".

```
>>> 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')
```

`dest` allows a custom attribute name to be provided:

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

Action classes

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to :meth:`add_argument`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1356); *backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1361)**
>
> Invalid class attribute value for "class" directive: "Action(option_strings, dest, nargs=None, const=None, default=None, \ type=None, choices=None, required=False, help=None, \ metavar=None)".
>
> ```
> .. class:: Action(option_strings, dest, nargs=None, const=None, default=None, \
>                   type=None, choices=None, required=False, help=None, \
>                   metavar=None)
> ```

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to :meth:`ArgumentParser.add_argument` except for the `action` itself.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1365); *backlink***
>
> Unknown interpreted text role "meth".

Instances of Action (or return value of any callable to the `action` parameter) should have attributes "dest", "option_strings", "default", "type", "required", "help", etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__`.

Action instances should be callable, so subclasses must override the `__call__` method, which should accept four parameters:

- `parser` - The ArgumentParser object which contains this action.
- `namespace` - The :class:`Namespace` object that will be returned by :meth:`~ArgumentParser.parse_args`. Most actions add an attribute to this object using :func:`setattr`.

  > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1381); *backlink***
  >
  > Unknown interpreted text role "class".

  > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1381); *backlink***
  >
  > Unknown interpreted text role "meth".

  > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1381); *backlink***
  >
  > Unknown interpreted text role "func".

- `values` - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the type keyword argument to :meth:`~ArgumentParser.add_argument`.

  > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1385); *backlink***
  >
  > Unknown interpreted text role "meth".

- `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.

The `__call__` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

Action subclasses can define a `format_usage` method that takes no argument and return a string which will be used when printing the usage of the program. If such method is not provided, a sensible default will be used.

The parse_args() method

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1403)**

Unknown directive type "method".

```
.. method:: ArgumentParser.parse_args(args=None, namespace=None)

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

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

   * args_ - List of strings to parse.  The default is taken from
     :data:`sys.argv`.

   * namespace_ - An object to take the attributes.  The default is a new empty
     :class:`Namespace` object.
```

Option value syntax

The :meth:`~ArgumentParser.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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1422); *backlink*

Unknown interpreted text role "meth".

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

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

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

For short options (options only one character long), the option and its value can be concatenated:

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

Several short options can be joined 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'])
Namespace(x=True, y=True, z='Z')
```

Invalid arguments

While parsing the command line, :meth:`~ArgumentParser.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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1461); *backlink*

Unknown interpreted text role "meth".

```
>>> 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
```

Arguments containing -

The :meth:`~ArgumentParser.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 argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The :meth:`~ArgumentParser.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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1489); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1489); *backlink*

Unknown interpreted text role "meth".

```
>>> 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 :meth:`~ArgumentParser.parse_args` that everything after that is a positional argument:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1527); *backlink*
>
> Unknown interpreted text role "meth".

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

Argument abbreviations (prefix matching)

The :meth:`~ArgumentParser.parse_args` method :ref:`by default <allow_abbrev>` allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1540); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1540); *backlink*
>
> Unknown interpreted text role "ref".

```
>>> 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
```

An error is produced for arguments that could produce more than one options. This feature can be disabled by setting :ref:`allow_abbrev` to `False`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1555); *backlink*
>
> Unknown interpreted text role "ref".

Beyond `sys.argv`

Sometimes it may be useful to have an ArgumentParser parse arguments other than those of :data:`sys.argv`. This can be accomplished by passing a list of strings to :meth:`~ArgumentParser.parse_args`. This is useful for testing at the interactive prompt:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1563); *backlink*
>
> Unknown interpreted text role "data".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1563); *backlink*
>
> Unknown interpreted text role "meth".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(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'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

The Namespace object

Simple class used by default by :meth:`~ArgumentParser.parse_args` to create an object holding attributes and return it.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1587); *backlink*
>
> Unknown interpreted text role "meth".

This class is deliberately simple, just an :class:`object` subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, :func:`vars`:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1590); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1590); backlink

Unknown interpreted text role "func".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an :class:`ArgumentParser` assign attributes to an already existing object, rather than a new :class:`Namespace` object. This can be achieved by specifying the `namespace=` keyword argument:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1600); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1600); backlink

Unknown interpreted text role "class".

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

Other utilities

Sub-commands

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1621)

Unknown directive type "method".

```
.. method:: ArgumentParser.add_subparsers([title], [description], [prog], \
                                          [parser_class], [action], \
                                          [option_string], [dest], [required], \
                                          [help], [metavar])

   Many 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``, and ``svn commit``.  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.
   :class:`ArgumentParser` supports the creation of such sub-commands with the
   :meth:`add_subparsers` method.  The :meth:`add_subparsers` method is normally
   called with no arguments and returns a special action object.  This object
   has a single method, :meth:`~ArgumentParser.add_parser`, which takes a
   command name and any :class:`ArgumentParser` constructor arguments, and
   returns an :class:`ArgumentParser` object that can be modified as usual.

   Description of parameters:

   * title - title for the sub-parser group in help output; by default
     "subcommands" if description is provided, otherwise uses title for
     positional arguments

   * description - description for the sub-parser group in help output, by
     default ``None``

   * prog - usage information that will be displayed with sub-command help,
     by default the name of the program and any positional arguments before the
     subparser argument

   * parser_class - class which will be used to create sub-parser instances, by
     default the class of the current parser (e.g. ArgumentParser)

   * action_ - the basic type of action to be taken when this argument is
     encountered at the command line

   * dest_ - name of the attribute under which sub-command name will be
     stored; by default ``None`` and no value is stored

   * required_ - Whether or not a subcommand must be provided, by default
     ``False`` (added in 3.7)

   * help_ - help for sub-parser group in help output, by default ``None``

   * metavar_ - string presenting available sub-commands in help; by default it
     is ``None`` and presents sub-commands in form {cmd1, cmd2, ..}

   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 argument 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 :meth:`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.  (A help message for each
subparser command, however, can be given by supplying the ``help=`` argument
to :meth:`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

  options:
    -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

  options:
    -h, --help  show this help message and exit

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

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

The :meth:`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} ...

  options:
    -h, --help  show this help message and exit

  subcommands:
    valid subcommands

    {foo,bar}   additional help

Furthermore, ``add_parser`` supports an additional ``aliases`` argument,
which allows multiple strings to refer to the same subparser. This example,
like ``svn``, aliases ``co`` as a shorthand for ``checkout``::

  >>> parser = argparse.ArgumentParser()
  >>> subparsers = parser.add_subparsers()
  >>> checkout = subparsers.add_parser('checkout', aliases=['co'])
  >>> checkout.add_argument('foo')
  >>> parser.parse_args(['co', 'bar'])
  Namespace(foo='bar')

One particularly effective way of handling sub-commands is to combine the use
of the :meth:`add_subparsers` method with calls to :meth:`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 :meth:`parse_args` do the job of calling the
appropriate function after 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 it is necessary
to check the name of the subparser that was invoked, the ``dest`` keyword
argument to the :meth:`add_subparsers` call will work::

  >>> 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')

.. versionchanged:: 3.7
   New *required* keyword argument.
```

FileType objects

The :class:`FileType` factory creates objects that can be passed to the type argument of :meth:`ArgumentParser.add_argument`. Arguments that have :class:`FileType` objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the :func:`open` function for more details):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1826); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1826); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1826); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1826); *backlink*

Unknown interpreted text role "func".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.FileIO name='raw.dat' mode='wb'>)
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1838); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1838); *backlink*

Unknown interpreted text role "class".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1847)

Unknown directive type "versionadded".

```
.. versionadded:: 3.4
   The *encodings* and *errors* keyword arguments.
```

Argument groups

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1854)

Unknown directive type "method".

```
.. method:: ArgumentParser.add_argument_group(title=None, description=None)

   By default, :class:`ArgumentParser` groups 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
   :meth:`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 :meth:`add_argument_group` method returns an argument group object which
   has an :meth:`~ArgumentParser.add_argument` method just like a regular
   :class:`ArgumentParser`.  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 :meth:`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.

.. versionchanged:: 3.11
 Calling :meth:`add_argument_group` on an argument group is deprecated.
 This feature was never supported and does not always work correctly.
 The function exists on the API by accident through inheritance and
 will be removed in the future.
```

Mutual exclusion

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1912)

Unknown directive type "method".

```
.. method:: ArgumentParser.add_mutually_exclusive_group(required=False)

   Create a mutually exclusive group. :mod:`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 :meth:`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
   :meth:`~ArgumentParser.add_argument_group`.

   .. versionchanged:: 3.11
    Calling :meth:`add_argument_group` or :meth:`add_mutually_exclusive_group`
    on a mutually exclusive group is deprecated. These features were never
    supported and do not always work correctly. The functions exist on the
    API by accident through inheritance and will be removed in the future.
```

Parser defaults

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1956)

Unknown directive type "method".

```
.. method:: ArgumentParser.set_defaults(**kwargs)

   Most of the time, the attributes of the object returned by :meth:`parse_args`
   will be fully determined by inspecting the command-line arguments and the argument
   actions.  :meth:`set_defaults` allows some additional
   attributes that are determined without any inspection of the command line to
   be added::

     >>> 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::

     >>> 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 working with multiple
   parsers.  See the :meth:`~ArgumentParser.add_subparsers` method for an
   example of this type.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1982)

Unknown directive type "method".

```
.. method:: ArgumentParser.get_default(dest)

   Get the default value for a namespace attribute, as set by either
```

```
:meth:`~ArgumentParser.add_argument` or by
:meth:`~ArgumentParser.set_defaults`::

  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', default='badger')
  >>> parser.get_default('foo')
  'badger'
```

Printing help

In most typical applications, :meth:`~ArgumentParser.parse_args` will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 1997); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2001)

Unknown directive type "method".

```
.. method:: ArgumentParser.print_usage(file=None)

   Print a brief description of how the :class:`ArgumentParser` should be
   invoked on the command line.  If *file* is ``None``, :data:`sys.stdout` is
   assumed.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2007)

Unknown directive type "method".

```
.. method:: ArgumentParser.print_help(file=None)

   Print a help message, including the program usage and information about the
   arguments registered with the :class:`ArgumentParser`.  If *file* is
   ``None``, :data:`sys.stdout` is assumed.
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2016)

Unknown directive type "method".

```
.. method:: ArgumentParser.format_usage()

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2021)

Unknown directive type "method".

```
.. method:: ArgumentParser.format_help()

   Return a string containing a help message, including the program usage and
   information about the arguments registered with the :class:`ArgumentParser`.
```

Partial parsing

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2030)

Unknown directive type "method".

```
.. method:: ArgumentParser.parse_known_args(args=None, namespace=None)
```

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 :meth:`~ArgumentParser.parse_known_args` method can be useful. It works much like :meth:`~ArgumentParser.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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2032); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2032); *backlink*

Unknown interpreted text role "meth".

```
>>> 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'])
```

Warning

:ref:`Prefix matching <prefix-matching>` rules apply to :meth:`parse_known_args`. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst,

line 2048); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2048); *backlink*

Unknown interpreted text role "meth".

Customizing file parsing

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2057)

Unknown directive type "method".

```
.. method:: ArgumentParser.convert_arg_line_to_args(arg_line)

   Arguments that are read from a file (see the *fromfile_prefix_chars*
   keyword argument to the :class:`ArgumentParser` constructor) are read one
   argument per line. :meth:`convert_arg_line_to_args` can be overridden for
   fancier reading.

   This method takes a single argument *arg_line* which is a string read from
   the argument file.  It returns a list of arguments parsed from this string.
   The method is called once per line read from the argument file, in order.

   A useful override of this method is one that treats each space-separated word
   as an argument.  The following example demonstrates how to do this::

    class MyArgumentParser(argparse.ArgumentParser):
        def convert_arg_line_to_args(self, arg_line):
            return arg_line.split()
```

Exiting methods

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2079)

Unknown directive type "method".

```
.. method:: ArgumentParser.exit(status=0, message=None)

   This method terminates the program, exiting with the specified *status*
   and, if given, it prints a *message* before that. The user can override
   this method to handle these steps differently::

    class ErrorCatchingArgumentParser(argparse.ArgumentParser):
        def exit(self, status=0, message=None):
            if status:
                raise Exception(f'Exiting because of an error: {message}')
            exit(status)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2091)

Unknown directive type "method".

```
.. method:: ArgumentParser.error(message)

   This method prints a usage message including the *message* to the
   standard error and terminates the program with a status code of 2.
```

Intermixed parsing

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2100)

Unknown directive type "method".

```
.. method:: ArgumentParser.parse_intermixed_args(args=None, namespace=None)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2101)

Unknown directive type "method".

```
.. method:: ArgumentParser.parse_known_intermixed_args(args=None, namespace=None)
```

A number of Unix commands allow the user to intermix optional arguments with positional arguments. The :meth:`~ArgumentParser.parse_intermixed_args` and :meth:`~ArgumentParser.parse_known_intermixed_args` methods support this parsing style.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2103); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2103); *backlink*

Unknown interpreted text role "meth".

These parsers do not support all the argparse features, and will raise exceptions if unsupported features are used. In particular, subparsers, `argparse.REMAINDER`, and mutually exclusive groups that include both optionals and positionals are not supported.

The following example shows the difference between :meth:`~ArgumentParser.parse_known_args` and :meth:`~ArgumentParser.parse_intermixed_args`: the former returns `['2', '3']` as unparsed arguments, while the latter collects all

the positionals into `rest`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2113); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2113); *backlink*

Unknown interpreted text role "meth".

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

:meth:`~ArgumentParser.parse_known_intermixed_args` returns a two item tuple containing the populated namespace and the list of remaining argument strings. :meth:`~ArgumentParser.parse_intermixed_args` raises an error if there are any remaining unparsed argument strings.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2128); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2128); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2133)

Unknown directive type "versionadded".

```
.. versionadded:: 3.7
```

Upgrading optparse code

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2140); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2140); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2140); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2140); *backlink*

Unknown interpreted text role "mod".

The :mod:`argparse` module improves on the standard library :mod:`optparse` module in a number of ways including:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2147); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2147); *backlink*

Unknown interpreted text role "mod".

- Handling positional arguments.
- Supporting sub-commands.
- Allowing alternative option prefixes like `+` and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

A partial upgrade path from :mod:`optparse` to :mod:`argparse`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2157); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)argparse.rst`, line 2157); *backlink*

Unknown interpreted text role "mod".

- Replace all :meth:`optparse.OptionParser.add_option` calls with :meth:`ArgumentParser.add_argument` calls.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2159); *backlink*

  Unknown interpreted text role "meth".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2159); *backlink*

  Unknown interpreted text role "meth".

- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional :meth:`ArgumentParser.add_argument` calls for the positional arguments. Keep in mind that what was previously called `options`, now in the :mod:`argparse` context is called `args`.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2162); *backlink*

  Unknown interpreted text role "meth".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2162); *backlink*

  Unknown interpreted text role "mod".

- Replace :meth:`optparse.OptionParser.disable_interspersed_args` by using :meth:`~ArgumentParser.parse_intermixed_args` instead of :meth:`~ArgumentParser.parse_args`.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2167); *backlink*

  Unknown interpreted text role "meth".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2167); *backlink*

  Unknown interpreted text role "meth".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2167); *backlink*

  Unknown interpreted text role "meth".

- 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 :class:`optparse.Values` with :class:`Namespace` and :exc:`optparse.OptionError` and :exc:`optparse.OptionValueError` with :exc:`ArgumentError`.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2177); *backlink*

  Unknown interpreted text role "class".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2177); *backlink*

  Unknown interpreted text role "class".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2177); *backlink*

  Unknown interpreted text role "exc".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2177); *backlink*

  Unknown interpreted text role "exc".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)argparse.rst, line 2177); *backlink*

  Unknown interpreted text role "exc".

- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.