Table of Contents

Pages

Tutorial

Why `OptionParser`?

When a Ruby program executes, it captures its command-line arguments and options into variable ARGV. This simple program just prints its `ARGV`:

```
p ARGV
```

Execution, with arguments and options:

```
$ ruby argv.rb foo --bar --baz bat bam
["foo", "--bar", "--baz", "bat", "bam"]
```

The executing program is responsible for parsing and handling the command-line options.

OptionParser offers methods for parsing and handling those options.

With `OptionParser`, you can define options so that for each option:

- The code that defines the option and code that handles that option are in the same place.

- The option may take no argument, a required argument, or an optional argument.
- The argument may be automatically converted to a specified class.
- The argument may be restricted to specified *forms*.
- The argument may be restricted to specified *values*.

The class also has method help, which displays automatically-generated help text.

Contents

To Begin With

To use `OptionParser`:

1. Require the `OptionParser` code.
2. Create an `OptionParser` object.
3. Define one or more options.
4. Parse the command line.

`File basic.rb` defines three options, `-x`, `-y`, and `-z`, each with a descriptive string, and each with a block.

```
# Require the OptionParser code.
require 'optparse'
# Create an OptionParser object.
parser = OptionParser.new
# Define one or more options.
parser.on('-x', 'Whether to X') do |value|
  p ['x', value]
end
parser.on('-y', 'Whether to Y') do |value|
  p ['y', value]
end
parser.on('-z', 'Whether to Z') do |value|
  p ['z', value]
end
# Parse the command line and return pared-down ARGV.
p parser.parse!
```

From these defined options, the parser automatically builds help text:

```
$ ruby basic.rb --help
Usage: basic [options]
    -x                               Whether to X
    -y                               Whether to Y
    -z                               Whether to Z
```

When an option is found during parsing, the block defined for the option is called with the argument value. An invalid option raises an exception.

`Method` parse!, which is used most often in this tutorial, removes from `ARGV` the options and arguments it finds, leaving other non-option arguments for the program to handle on its own. The method returns the possibly-reduced `ARGV` array.

Executions:

```
$ ruby basic.rb -x -z
["x", true]
["z", true]
[]
$ ruby basic.rb -z -y -x
["z", true]
["y", true]
["x", true]
[]
$ ruby basic.rb -x input_file.txt output_file.txt
["x", true]
["input_file.txt", "output_file.txt"]
$ ruby basic.rb -a
basic.rb:16:in `<main>': invalid option: -a (OptionParser::InvalidOption)
```

Defining Options

A common way to define an option in `OptionParser` is with instance method OptionParser#on.

The method may be called with any number of arguments (whose order does not matter), and may also have a trailing optional keyword argument `into`.

The given arguments determine the characteristics of the new option. These may include:

- One or more short option names.
- One or more long option names.
- Whether the option takes no argument, an optional argument, or a required argument.
- Acceptable *forms* for the argument.
- Acceptable *values* for the argument.
- A proc or method to be called when the parser encounters the option.
- `String` descriptions for the option.

Option Names

You can give an option one or more names of two types:

- Short (1-character) name, beginning with one hyphen (`-`).

- Long (multi-character) name, beginning with two hyphens (`--`).

Short Option Names

A short option name consists of a hyphen and a single character.

`File short_names.rb` defines an option with a short name, `-x`, and an option with two short names (aliases, in effect) `-y` and `-z`.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x', 'Short name') do |value|
  p ['x', value]
end
parser.on('-1', '-%', 'Two short names') do |value|
  p ['-1 or -%', value]
end
parser.parse!
```

Executions:

```
$ ruby short_names.rb --help
Usage: short_names [options]
    -x                               Short name
    -1, -%                           Two short names
$ ruby short_names.rb -x
["x", true]
$ ruby short_names.rb -1
["-1 or -%", true]
$ ruby short_names.rb -%
["-1 or -%", true]
```

Multiple short names can “share” a hyphen:

```
$ ruby short_names.rb -x1%
["x", true]
["-1 or -%", true]
["-1 or -%", true]
```

Long Option Names

A long option name consists of two hyphens and a one or more characters (usually two or more characters).

`File long_names.rb` defines an option with a long name, `--xxx`, and an option with two long names (aliases, in effect) `--y1%` and `--z2#`.

```
require 'optparse'
parser = OptionParser.new
parser.on('--xxx', 'Long name') do |value|
  p ['-xxx', value]
end
parser.on('--y1%', '--z2#', "Two long names") do |value|
  p ['--y1% or --z2#', value]
```

```
end
parser.parse!
```

Executions:

```
$ ruby long_names.rb --help
Usage: long_names [options]
        --xxx                        Long name
        --y1%, --z2#                 Two long names
$ ruby long_names.rb --xxx
["-xxx", true]
$ ruby long_names.rb --y1%
["--y1% or --z2#", true]
$ ruby long_names.rb --z2#
["--y1% or --z2#", true]
```

A long name may be defined with both positive and negative senses.

File `long_with_negation.rb` defines an option that has both senses.

```
require 'optparse'
parser = OptionParser.new
parser.on('--[no-]binary', 'Long name with negation') do |value|
  p [value, value.class]
end
parser.parse!
```

Executions:

```
$ ruby long_with_negation.rb --help
Usage: long_with_negation [options]
        --[no-]binary                Long name with negation
$ ruby long_with_negation.rb --binary
[true, TrueClass]
$ ruby long_with_negation.rb --no-binary
[false, FalseClass]
```

Mixing Option Names

Many developers like to mix short and long option names, so that a short name is in effect an abbreviation of a long name.

File `mixed_names.rb` defines options that each have both a short and a long name.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x', '--xxx', 'Short and long, no argument') do |value|
  p ['--xxx', value]
end
parser.on('-yYYY', '--yyy', 'Short and long, required argument') do |value|
  p ['--yyy', value]
end
parser.on('-z [ZZZ]', '--zzz', 'Short and long, optional argument') do |value
```

```
  p ['--zzz', value]
end
parser.parse!
```

Executions:

```
$ ruby mixed_names.rb --help
Usage: mixed_names [options]
    -x, --xxx                        Short and long, no argument
    -y, --yyyYYY                     Short and long, required argument
    -z, --zzz [ZZZ]                  Short and long, optional argument
$ ruby mixed_names.rb -x
["--xxx", true]
$ ruby mixed_names.rb --xxx
["--xxx", true]
$ ruby mixed_names.rb -y
mixed_names.rb:12:in `<main>': missing argument: -y (OptionParser::MissingArg
$ ruby mixed_names.rb -y FOO
["--yyy", "FOO"]
$ ruby mixed_names.rb --yyy
mixed_names.rb:12:in `<main>': missing argument: --yyy (OptionParser::Missing
$ ruby mixed_names.rb --yyy BAR
["--yyy", "BAR"]
$ ruby mixed_names.rb -z
["--zzz", nil]
$ ruby mixed_names.rb -z BAZ
["--zzz", "BAZ"]
$ ruby mixed_names.rb --zzz
["--zzz", nil]
$ ruby mixed_names.rb --zzz BAT
["--zzz", "BAT"]
```

Option Name Abbreviations

By default, abbreviated option names on the command-line are allowed. An abbreviated name is valid if it is unique among abbreviated option names.

```
require 'optparse'
parser = OptionParser.new
parser.on('-n', '--dry-run',) do |value|
  p ['--dry-run', value]
end
parser.on('-d', '--draft',) do |value|
  p ['--draft', value]
end
parser.parse!
```

Executions:

```
$ ruby name_abbrev.rb --help
Usage: name_abbrev [options]
    -n, --dry-run
    -d, --draft
$ ruby name_abbrev.rb -n
["--dry-run", true]
```

```
$ ruby name_abbrev.rb --dry-run
["--dry-run", true]
$ ruby name_abbrev.rb -d
["--draft", true]
$ ruby name_abbrev.rb --draft
["--draft", true]
$ ruby name_abbrev.rb --d
name_abbrev.rb:9:in `<main>': ambiguous option: --d (OptionParser::AmbiguousO
$ ruby name_abbrev.rb --dr
name_abbrev.rb:9:in `<main>': ambiguous option: --dr (OptionParser::Ambiguous
$ ruby name_abbrev.rb --dry
["--dry-run", true]
$ ruby name_abbrev.rb --dra
["--draft", true]
```

You can disable abbreviation using method `require_exact`.

```
require 'optparse'
parser = OptionParser.new
parser.on('-n', '--dry-run',) do |value|
  p ['--dry-run', value]
end
parser.on('-d', '--draft',) do |value|
  p ['--draft', value]
end
parser.require_exact = true
parser.parse!
```

Executions:

```
$ ruby no_abbreviation.rb --dry-ru
no_abbreviation.rb:10:in `<main>': invalid option: --dry-ru (OptionParser::In
$ ruby no_abbreviation.rb --dry-run
["--dry-run", true]
```

Option Arguments

An option may take no argument, a required argument, or an optional argument.

Option with No Argument

All the examples above define options with no argument.

Option with Required Argument

Specify a required argument for an option by adding a dummy word to its name definition.

`File required_argument.rb` defines two options; each has a required argument because the name definition has a following dummy word.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x XXX', '--xxx', 'Required argument via short name') do |value|
  p ['--xxx', value]
end
parser.on('-y', '--y YYY', 'Required argument via long name') do |value|
  p ['--yyy', value]
end
parser.parse!
```

When an option is found, the given argument is yielded.

Executions:

```
$ ruby required_argument.rb --help
Usage: required_argument [options]
    -x, --xxx XXX                    Required argument via short name
    -y, --y YYY                      Required argument via long name
$ ruby required_argument.rb -x AAA
["--xxx", "AAA"]
$ ruby required_argument.rb -y BBB
["--yyy", "BBB"]
```

Omitting a required argument raises an error:

```
$ ruby required_argument.rb -x
required_argument.rb:9:in `<main>': missing argument: -x (OptionParser::Missi
```

Option with Optional Argument

Specify an optional argument for an option by adding a dummy word enclosed in square brackets to its name definition.

`File optional_argument.rb` defines two options; each has an optional argument because the name definition has a following dummy word in square brackets.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x [XXX]', '--xxx', 'Optional argument via short  name') do |value
  p ['--xxx', value]
end
parser.on('-y', '--yyy [YYY]', 'Optional argument via long name') do |value|
  p ['--yyy', value]
end
parser.parse!
```

When an option with an argument is found, the given argument yielded.

Executions:

```
$ ruby optional_argument.rb --help
Usage: optional_argument [options]
```

```
    -x, --xxx [XXX]                  Optional argument via short  name
    -y, --yyy [YYY]                  Optional argument via long name
$ ruby optional_argument.rb -x AAA
["--xxx", "AAA"]
$ ruby optional_argument.rb -y BBB
["--yyy", "BBB"]
```

Omitting an optional argument does not raise an error.

Argument Values

Permissible argument values may be restricted either by specifying explicit values or by providing a pattern that the given value must match.

Explicit Argument Values

You can specify argument values in either of two ways:

- Specify values an array of strings.
- Specify values a hash.

Explicit Values in `Array`

You can specify explicit argument values in an array of strings. The argument value must be one of those strings, or an unambiguous abbreviation.

`File explicit_array_values.rb` defines options with explicit argument values.

```
require 'optparse'
parser = OptionParser.new
parser.on('-xXXX', ['foo', 'bar'], 'Values for required argument' ) do |value
  p ['-x', value]
end
parser.on('-y [YYY]', ['baz', 'bat'], 'Values for optional argument') do |val
  p ['-y', value]
end
parser.parse!
```

Executions:

```
$ ruby explicit_array_values.rb --help
Usage: explicit_array_values [options]
    -xXXX                            Values for required argument
    -y [YYY]                         Values for optional argument
$ ruby explicit_array_values.rb -x
explicit_array_values.rb:9:in `<main>': missing argument: -x (OptionParser::M
$ ruby explicit_array_values.rb -x foo
["-x", "foo"]
$ ruby explicit_array_values.rb -x f
["-x", "foo"]
$ ruby explicit_array_values.rb -x bar
["-x", "bar"]
```

```
$ ruby explicit_array_values.rb -y ba
explicit_array_values.rb:9:in `<main>': ambiguous argument: -y ba (OptionPars
$ ruby explicit_array_values.rb -x baz
explicit_array_values.rb:9:in `<main>': invalid argument: -x baz (OptionParse
```

Explicit Values in Hash

You can specify explicit argument values in a hash with string keys. The value passed must be one of those keys, or an unambiguous abbreviation; the value yielded will be the value for that key.

File `explicit_hash_values.rb` defines options with explicit argument values.

```
require 'optparse'
parser = OptionParser.new
parser.on('-xXXX', {foo: 0, bar: 1}, 'Values for required argument' ) do |val
  p ['-x', value]
end
parser.on('-y [YYY]', {baz: 2, bat: 3}, 'Values for optional argument') do |v
  p ['-y', value]
end
parser.parse!
```

Executions:

```
$ ruby explicit_hash_values.rb --help
Usage: explicit_hash_values [options]
    -xXXX                            Values for required argument
    -y [YYY]                         Values for optional argument
$ ruby explicit_hash_values.rb -x
explicit_hash_values.rb:9:in `<main>': missing argument: -x (OptionParser::Mi
$ ruby explicit_hash_values.rb -x foo
["-x", 0]
$ ruby explicit_hash_values.rb -x f
["-x", 0]
$ ruby explicit_hash_values.rb -x bar
["-x", 1]
$ ruby explicit_hash_values.rb -x baz
explicit_hash_values.rb:9:in `<main>': invalid argument: -x baz (OptionParser
$ ruby explicit_hash_values.rb -y
["-y", nil]
$ ruby explicit_hash_values.rb -y baz
["-y", 2]
$ ruby explicit_hash_values.rb -y bat
["-y", 3]
$ ruby explicit_hash_values.rb -y ba
explicit_hash_values.rb:9:in `<main>': ambiguous argument: -y ba (OptionParse
$ ruby explicit_hash_values.rb -y bam
["-y", nil]
```

Argument Value Patterns

You can restrict permissible argument values by specifying a `Regexp` that the given argument must match.

`File matched_values.rb` defines options with matched argument values.

```
require 'optparse'
parser = OptionParser.new
parser.on('--xxx XXX', /foo/i, 'Matched values') do |value|
  p ['--xxx', value]
end
parser.parse!
```

Executions:

```
$ ruby matched_values.rb --help
Usage: matched_values [options]
        --xxx XXX                    Matched values
$ ruby matched_values.rb --xxx foo
["--xxx", "foo"]
$ ruby matched_values.rb --xxx FOO
["--xxx", "FOO"]
$ ruby matched_values.rb --xxx bar
matched_values.rb:6:in `<main>': invalid argument: --xxx bar (OptionParser::I
```

Keyword Argument `into`

In parsing options, you can add keyword option `into` with a hash-like argument; each parsed option will be added as a name/value pair.

This is useful for:

- Collecting options.
- Checking for missing options.
- Providing default values for options.

Collecting Options

Use keyword argument `into` to collect options.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x', '--xxx', 'Short and long, no argument')
parser.on('-yYYY', '--yyy', 'Short and long, required argument')
parser.on('-z [ZZZ]', '--zzz', 'Short and long, optional argument')
options = {}
parser.parse!(into: options)
p options
```

Executions:

```
$ ruby collected_options.rb --help
Usage: into [options]
    -x, --xxx                        Short and long, no argument
    -y, --yyyYYY                     Short and long, required argument
    -z, --zzz [ZZZ]                  Short and long, optional argument
$ ruby collected_options.rb --xxx
{:xxx=>true}
$ ruby collected_options.rb --xxx --yyy FOO
{:xxx=>true, :yyy=>"FOO"}
$ ruby collected_options.rb --xxx --yyy FOO --zzz Bar
{:xxx=>true, :yyy=>"FOO", :zzz=>"Bar"}
$ ruby collected_options.rb --xxx --yyy FOO --yyy BAR
{:xxx=>true, :yyy=>"BAR"}
```

Note in the last execution that the argument value for option `--yyy` was overwritten.

Checking for Missing Options

Use the collected options to check for missing options.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x', '--xxx', 'Short and long, no argument')
parser.on('-yYYY', '--yyy', 'Short and long, required argument')
parser.on('-z [ZZZ]', '--zzz', 'Short and long, optional argument')
options = {}
parser.parse!(into: options)
required_options = [:xxx, :zzz]
missing_options = required_options - options.keys
unless missing_options.empty?
  fail "Missing required options: #{missing_options}"
end
```

Executions:

```
$ ruby missing_options.rb --help
Usage: missing_options [options]
    -x, --xxx                        Short and long, no argument
    -y, --yyyYYY                     Short and long, required argument
    -z, --zzz [ZZZ]                  Short and long, optional argument
$ ruby missing_options.rb --yyy FOO
missing_options.rb:11:in `<main>': Missing required options: [:xxx, :zzz] (Ru
```

Default Values for Options

Initialize the `into` argument to define default values for options.

```
require 'optparse'
parser = OptionParser.new
parser.on('-x', '--xxx', 'Short and long, no argument')
parser.on('-yYYY', '--yyy', 'Short and long, required argument')
parser.on('-z [ZZZ]', '--zzz', 'Short and long, optional argument')
options = {yyy: 'AAA', zzz: 'BBB'}
```

```
parser.parse!(into: options)
p options
```

Executions:

```
$ ruby default_values.rb --help
Usage: default_values [options]
    -x, --xxx                        Short and long, no argument
    -y, --yyyYYY                     Short and long, required argument
    -z, --zzz [ZZZ]                  Short and long, optional argument
$ ruby default_values.rb --yyy FOO
{:yyy=>"FOO", :zzz=>"BBB"}
```

Argument Converters

An option can specify that its argument is to be converted from the default `String` to an instance of another class. There are a number of built-in converters.

Example: `File` `date.rb` defines an option whose argument is to be converted to a `Date` object. The argument is converted by method Date#parse.

```
require 'optparse/date'
parser = OptionParser.new
parser.on('--date=DATE', Date) do |value|
  p [value, value.class]
end
parser.parse!
```

Executions:

```
$ ruby date.rb --date 2001-02-03
[#<Date: 2001-02-03 ((2451944j,0s,0n),+0s,2299161j)>, Date]
$ ruby date.rb --date 20010203
[#<Date: 2001-02-03 ((2451944j,0s,0n),+0s,2299161j)>, Date]
$ ruby date.rb --date "3rd Feb 2001"
[#<Date: 2001-02-03 ((2451944j,0s,0n),+0s,2299161j)>, Date]
```

You can also define custom converters. See Argument Converters for both built-in and custom converters.

Help

`OptionParser` makes automatically generated help text available.

The help text consists of:

- A banner, showing the usage.
- Option short and long names.
- Option dummy argument names.

- Option descriptions.

Example code:

```
require 'optparse'
parser = OptionParser.new
parser.on(
  '-x', '--xxx',
  'Adipiscing elit. Aenean commodo ligula eget.',
  'Aenean massa. Cum sociis natoque penatibus',
  )
parser.on(
  '-y', '--yyy YYY',
  'Lorem ipsum dolor sit amet, consectetuer.'
)
parser.on(
  '-z', '--zzz [ZZZ]',
  'Et magnis dis parturient montes, nascetur',
  'ridiculus mus. Donec quam felis, ultricies',
  'nec, pellentesque eu, pretium quis, sem.',
  )
parser.parse!
```

The option names and dummy argument names are defined as described above.

The option description consists of the strings that are not themselves option names; An option can have more than one description string. Execution:

```
Usage: help [options]
    -x, --xxx                        Adipiscing elit. Aenean commodo ligula e
                                     Aenean massa. Cum sociis natoque penatib
    -y, --yyy YYY                    Lorem ipsum dolor sit amet, consectetuer
    -z, --zzz [ZZZ]                  Et magnis dis parturient montes, nascetu
                                     ridiculus mus. Donec quam felis, ultrici
                                     nec, pellentesque eu, pretium quis, sem.
```

The program name is included in the default banner: `Usage: #{program_name} [options]` ; you can change the program name.

```
require 'optparse'
parser = OptionParser.new
parser.program_name = 'help_program_name.rb'
parser.parse!
```

Execution:

```
$ ruby help_program_name.rb --help
Usage: help_program_name.rb [options]
```

You can also change the entire banner.

```
require 'optparse'
parser = OptionParser.new
```

```
parser.banner = "Usage: ruby help_banner.rb"
parser.parse!
```

Execution:

```
$ ruby help_banner.rb --help
Usage: ruby help_banner.rb
```

By default, the option names are indented 4 spaces and the width of the option-names field is 32 spaces.

You can change these values, along with the banner, by passing parameters to OptionParser.new.

```
require 'optparse'
parser = OptionParser.new(
  'ruby help_format.rb [options]', # Banner
  20,                               # Width of options field
  ' ' * 2                               # Indentation
)
parser.on(
  '-x', '--xxx',
  'Adipiscing elit. Aenean commodo ligula eget.',
  'Aenean massa. Cum sociis natoque penatibus',
  )
parser.on(
  '-y', '--yyy YYY',
  'Lorem ipsum dolor sit amet, consectetuer.'
)
parser.on(
  '-z', '--zzz [ZZZ]',
  'Et magnis dis parturient montes, nascetur',
  'ridiculus mus. Donec quam felis, ultricies',
  'nec, pellentesque eu, pretium quis, sem.',
  )
parser.parse!
```

Execution:

```
$ ruby help_format.rb --help
ruby help_format.rb [options]
  -x, --xxx            Adipiscing elit. Aenean commodo ligula eget.
                       Aenean massa. Cum sociis natoque penatibus
  -y, --yyy YYY        Lorem ipsum dolor sit amet, consectetuer.
  -z, --zzz [ZZZ]      Et magnis dis parturient montes, nascetur
                       ridiculus mus. Donec quam felis, ultricies
                       nec, pellentesque eu, pretium quis, sem.
```

Top List and Base List

An `OptionParser` object maintains a stack of OptionParser::List objects, each of which has a collection of zero or more options. It is unlikely that you'll need to add or take away from that stack.

The stack includes:

- The *top list*, given by OptionParser#top.
- The *base list*, given by OptionParser#base.

When `OptionParser` builds its help text, the options in the top list precede those in the base list.

Methods for Defining Options

Option-defining methods allow you to create an option, and also append/prepend it to the top list or append it to the base list.

Each of these next three methods accepts a sequence of parameter arguments and a block, creates an option object using method OptionParser#make_switch (see below), and returns the created option:

- Method OptionParser#define appends the created option to the top list.
- Method OptionParser#define_head prepends the created option to the top list.
- Method OptionParser#define_tail appends the created option to the base list.

These next three methods are identical to the three above, except for their return values:

- Method OptionParser#on is identical to method OptionParser#define, except that it returns the parser object `self`.
- Method OptionParser#on_head is identical to method OptionParser#define_head, except that it returns the parser object `self`.
- Method OptionParser#on_tail is identical to method OptionParser#define_tail, except that it returns the parser object `self`.

Though you may never need to call it directly, here's the core method for defining an option:

- Method OptionParser#make_switch accepts an array of parameters and a block. See Parameters for New Options. This method is unlike others here in that it:
  - Accepts an *array of parameters*; others accept a *sequence of parameter arguments*.
  - Returns an array containing the created option object, option names, and other values; others return either the created option object or the parser object `self`.

Parsing

`OptionParser` has six instance methods for parsing.

Three have names ending with a “bang” (`!`):

- parse!
- order!
- permute!

Each of these methods:

- Accepts an optional array of string arguments `argv` ; if not given, `argv` defaults to the value of OptionParser#default_argv, whose initial value is ARGV.
- Accepts an optional keyword argument `into` (see Keyword Argument into).
- Returns `argv` , possibly with some elements removed.

The three other methods have names *not* ending with a “bang”:

- parse
- order
- permute

Each of these methods:

- Accepts an array of string arguments *or* zero or more string arguments.
- Accepts an optional keyword argument `into` and its value *into*. (see Keyword Argument into).
- Returns `argv` , possibly with some elements removed.

Method `parse!`

Method `parse!` :

- Accepts an optional array of string arguments `argv` ; if not given, `argv` defaults to the value of OptionParser#default_argv, whose initial value is ARGV.
- Accepts an optional keyword argument `into` (see Keyword Argument into).
- Returns `argv` , possibly with some elements removed.

The method processes the elements in `argv` beginning at `argv[0]` , and ending, by default, at the end.

Otherwise processing ends and the method returns when:

- The terminator argument `--` is found; the terminator argument is removed before the return.

- Environment variable `POSIXLY_CORRECT` is defined and a non-option argument is found; the non-option argument is not removed. Note that the *value* of that variable does not matter, as only its existence is checked.

File `parse_bang.rb`:

```
require 'optparse'
parser = OptionParser.new
parser.on('--xxx') do |value|
  p ['--xxx', value]
end
parser.on('--yyy YYY') do |value|
  p ['--yyy', value]
end
parser.on('--zzz [ZZZ]') do |value|
  p ['--zzz', value]
end
ret = parser.parse!
puts "Returned: #{ret} (#{ret.class})"
```

Help:

```
$ ruby parse_bang.rb --help
Usage: parse_bang [options]
        --xxx
        --yyy YYY
        --zzz [ZZZ]
```

Default behavior:

```
$ ruby parse_bang.rb input_file.txt output_file.txt --xxx --yyy FOO --zzz BAR
["--xxx", true]
["--yyy", "FOO"]
["--zzz", "BAR"]
Returned: ["input_file.txt", "output_file.txt"] (Array)
```

Processing ended by terminator argument:

```
$ ruby parse_bang.rb input_file.txt output_file.txt --xxx --yyy FOO -- --zzz
["--xxx", true]
["--yyy", "FOO"]
Returned: ["input_file.txt", "output_file.txt", "--zzz", "BAR"] (Array)
```

Processing ended by non-option found when `POSIXLY_CORRECT` is defined:

```
$ POSIXLY_CORRECT=true ruby parse_bang.rb --xxx input_file.txt output_file.tx
["--xxx", true]
Returned: ["input_file.txt", "output_file.txt", "-yyy", "FOO"] (Array)
```

Method parse

Method `parse`:

- Accepts an array of string arguments *or* zero or more string arguments.
- Accepts an optional keyword argument `into` and its value *into*. (see Keyword Argument into).
- Returns `argv`, possibly with some elements removed.

If given an array `ary`, the method forms array `argv` as `ary.dup`. If given zero or more string arguments, those arguments are formed into array `argv`.

The method calls

```
parse!(argv, into: into)
```

Note that environment variable `POSIXLY_CORRECT` and the terminator argument `--` are honored.

`File` `parse.rb`:

```
require 'optparse'
parser = OptionParser.new
parser.on('--xxx') do |value|
  p ['--xxx', value]
end
parser.on('--yyy YYY') do |value|
  p ['--yyy', value]
end
parser.on('--zzz [ZZZ]') do |value|
  p ['--zzz', value]
end
ret = parser.parse(ARGV)
puts "Returned: #{ret} (#{ret.class})"
```

Help:

```
$ ruby parse.rb --help
Usage: parse [options]
        --xxx
        --yyy YYY
        --zzz [ZZZ]
```

Default behavior:

```
$ ruby parse.rb input_file.txt output_file.txt --xxx --yyy FOO --zzz BAR
["--xxx", true]
["--yyy", "FOO"]
["--zzz", "BAR"]
Returned: ["input_file.txt", "output_file.txt"] (Array)
```

Processing ended by terminator argument:

```
$ ruby parse.rb input_file.txt output_file.txt --xxx --yyy FOO -- --zzz BAR
["--xxx", true]
["--yyy", "FOO"]
Returned: ["input_file.txt", "output_file.txt", "--zzz", "BAR"] (Array)
```

Processing ended by non-option found when `POSIXLY_CORRECT` is defined:

```
$ POSIXLY_CORRECT=true ruby parse.rb --xxx input_file.txt output_file.txt -yy
["--xxx", true]
Returned: ["input_file.txt", "output_file.txt", "-yyy", "FOO"] (Array)
```

Method `order!`

Calling method OptionParser#order! gives exactly the same result as calling method OptionParser#parse! with environment variable `POSIXLY_CORRECT` defined.

Method `order`

Calling method OptionParser#order gives exactly the same result as calling method OptionParser#parse with environment variable `POSIXLY_CORRECT` defined.

Method `permute!`

Calling method OptionParser#permute! gives exactly the same result as calling method OptionParser#parse! with environment variable `POSIXLY_CORRECT` *not* defined.

Method `permute`

Calling method OptionParser#permute gives exactly the same result as calling method OptionParser#parse with environment variable `POSIXLY_CORRECT` *not* defined.