argparse

Build Status

CLI arguments parser for node.js. Javascript port of python's argparse module (original version 3.2). That's a full port, except some very rare options, recorded in issue tracker.

NB. Difference with original.

Method names changed to camelCase. See generated docs.

Use `defaultValue` instead of `default`.

Use `argparse.Const.REMAINDER` instead of `argparse.REMAINDER`, and similarly for constant values `OPTIONAL`, `ZERO_OR_MORE`, and `ONE_OR_MORE` (aliases for `nargs` values `'?'`, `'*'`, `'+'`, respectively), and `SUPPRESS`.

Example

test.js file:

```
#!/usr/bin/env node
'use strict';

var ArgumentParser = require('../lib/
        argparse').ArgumentParser;
var parser = new ArgumentParser({
  version: '0.0.1',
  addHelp:true,
  description: 'Argparse example'
});
parser.addArgument(
  [ '-f', '--foo' ],
  {
    help: 'foo bar'
  }
);
parser.addArgument(
  [ '-b', '--bar' ],
  {
    help: 'bar foo'
  }
);
parser.addArgument(
  '--baz',
  {
    help: 'baz bar'
  }
```

```
);
var args = parser.parseArgs();
console.dir(args);
```

Display help:

```
$ ./test.js -h
usage: example.js [-h] [-v] [-f FOO] [-b
BAR] [--baz BAZ]

Argparse example

Optional arguments:
  -h, --help            Show this help
message and exit.
  -v, --version         Show program's
version number and exit.
  -f FOO, --foo FOO     foo bar
  -b BAR, --bar BAR     bar foo
  --baz BAZ             baz bar
```

Parse arguments:

```
$ ./test.js -f=3 --bar=4 --baz 5
{ foo: '3', bar: '4', baz: '5' }
```

More examples.

ArgumentParser objects

```
new ArgumentParser({parameters hash});
```

Creates a new ArgumentParser object.

Supported params:

- `description` - Text to display before the argument help.
- `epilog` - Text to display after the argument help.
- `addHelp` - Add a -h/–help option to the parser. (default: true)
- `argumentDefault` - Set the global default value for arguments. (default: null)
- `parents` - A list of ArgumentParser objects whose arguments should also be included.
- `prefixChars` - The set of characters that prefix optional arguments. (default: ‘-‘)
- `formatterClass` - A class for customizing the help output.
- `prog` - The name of the program (default: `path.basename(process.argv[1])`)
- `usage` - The string describing the program usage (default: generated)
- `conflictHandler` - Usually unnecessary, defines strategy for resolving conflicting optionals.

Not supported yet

- `fromfilePrefixChars` - The set of characters that prefix files from which additional arguments should be read.

Details in original ArgumentParser guide

License

Contributors

- Eugene Shkuropat
- Paul Jacobson

  others

addArgument() method

```
ArgumentParser.addArgument(name or flag
or [name] or [flags...], {options})
```

Defines how a single command-line argument should be parsed.

- `name or flag or [name] or [flags...]` - Either a positional name (e.g., `'foo'`), a single option (e.g., `'-f'` or `'--foo'`), an array of a single positional name (e.g., `['foo']`), or an array of options (e.g., `['-f', '--foo']`).

Options:

- `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.
- `constant` - A constant value required by some action and nargs selections.
- `defaultValue` - The value produced if the argument is absent from the command line.
- `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 parseArgs().

Details in original add_argument guide

Details in original sub-commands guide

```
  description: 'Argparse examples: sub-
        commands',
});

var subparsers = parser.addSubparsers({
  title:'subcommands',
  dest:"subcommand_name"
});

var bar = subparsers.addParser('c1',
        {addHelp:true});
bar.addArgument(
  [ '-f', '--foo' ],
  {
    action: 'store',
    help: 'foo3 bar3'
  }
);
var bar = subparsers.addParser(
  'c2',
  {aliases:['co'], addHelp:true}
);
bar.addArgument(
  [ '-b', '--bar' ],
  {
    action: 'store',
    type: 'int',
    help: 'foo3 bar3'
  }
);

var args = parser.parseArgs();
console.dir(args);
```


Action (some details)

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 parseArgs(). The action keyword argument specifies how the command-line arguments should be handled. The supported actions are:

- `store` - Just stores the argument's value. This is the default action.
- `storeConst` - Stores value, specified by the const keyword argument. (Note that the const keyword argument defaults to the rather unhelpful None.) The 'storeConst' action is most commonly used with optional arguments, that specify some sort of flag.
- `storeTrue` and `storeFalse` - Stores values True and False respectively. These are special cases of 'storeConst'.
- `append` - Stores a list, and appends each argument value to the list. This is useful to allow an option to be specified multiple times.
- `appendConst` - Stores a list, and appends value, specified by the const keyword argument to the list. (Note, that the const keyword argument defaults is None.) The 'appendConst' action is typically used when multiple arguments need to store constants to the same list.

- `count` - Counts the number of times a keyword argument occurs. For example, used for increasing verbosity levels.
- `help` - 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 ArgumentParser for details of how the output is created.
- `version` - Prints version information and exit. Expects a `version=` keyword argument in the addArgument() call.

Details in original action guide

Sub-commands

ArgumentParser.addSubparsers()

Many programs split 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. `ArgumentParser` supports creation of such sub-commands with `addSubparsers()` method. The `addSubparsers()` method is normally called with no arguments and returns an special action object. This object has a single method `addParser()`, which takes a command name and any `ArgumentParser` constructor arguments, and returns an `ArgumentParser` object that can be modified as usual.

Example:

sub_commands.js

```
#!/usr/bin/env node
'use strict';

var ArgumentParser = require('../lib/
        argparse').ArgumentParser;
var parser = new ArgumentParser({
  version: '0.0.1',
  addHelp:true,
```