Description

pflag is a drop-in replacement for Go's flag package, implementing POSIX/GNU-style --flags.

pflag is compatible with the GNU extensions to the POSIX recommendations for command-line options. For a more precise description, see the "Command-line flag syntax" section below.

pflag is available under the same style of BSD license as the Go language, which can be found in the LICENSE file.

Installation

pflag is available using the standard `go get` command.

Install by running:

```
go get github.com/spf13/pflag
```

Run tests by running:

```
go test github.com/spf13/pflag
```

Usage

pflag is a drop-in replacement of Go's native flag package. If you import pflag under the name "flag" then all code should continue to function with no changes.

```
import flag "github.com/spf13/pflag"
```

There is one exception to this: if you directly instantiate the Flag struct there is one more field "Shorthand" that you will need to set. Most code never instantiates this struct directly, and instead uses functions such as String(), BoolVar(), and Var(), and is therefore unaffected.

Define flags using flag.String(), Bool(), Int(), etc.

This declares an integer flag, -flagname, stored in the pointer ip, with type *int.

```
var ip *int = flag.Int("flagname", 1234, "help message for flagname")
```

If you like, you can bind the flag to a variable using the Var() functions.

```
var flagvar int
func init() {
    flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")
}
```

Or you can create custom flags that satisfy the Value interface (with pointer receivers) and couple them to flag parsing by

```
flag.Var(&flagVal, "name", "help message for flagname")
```

For such flags, the default value is just the initial value of the variable.

After all flags are defined, call

```
flag.Parse()
```

to parse the command line into the defined flags.

Flags may then be used directly. If you're using the flags themselves, they are all pointers; if you bind to variables, they're values.

```
fmt.Println("ip has value ", *ip)
fmt.Println("flagvar has value ", flagvar)
```

There are helper functions available to get the value stored in a Flag if you have a FlagSet but find it difficult to keep up with all of the pointers in your code. If you have a pflag.FlagSet with a flag called 'flagname' of type int you can use GetInt() to get the int value. But notice that 'flagname' must exist and it must be an int. GetString("flagname") will fail.

```
i, err := flagset.GetInt("flagname")
```

After parsing, the arguments after the flag are available as the slice flag.Args() or individually as flag.Arg(i). The arguments are indexed from 0 through flag.NArg()-1.

The pflag package also defines some new functions that are not in flag, that give one-letter shorthands for flags. You can use these by appending 'P' to the name of any function that defines a flag.

```
var ip = flag.IntP("flagname", "f", 1234, "help message")
var flagvar bool
func init() {
    flag.BoolVarP(&flagvar, "boolname", "b", true, "help message")
}
flag.VarP(&flagVal, "varname", "v", "help message")
```

Shorthand letters can be used with single dashes on the command line. Boolean shorthand flags can be combined with other shorthand flags.

The default set of command-line flags is controlled by top-level functions. The FlagSet type allows one to define independent sets of flags, such as to implement subcommands in a command-line interface. The methods of FlagSet are analogous to the top-level functions for the command-line flag set.

Setting no option default values for flags

After you create a flag it is possible to set the pflag.NoOptDefVal for the given flag. Doing this changes the meaning of the flag slightly. If a flag has a NoOptDefVal and the flag is set on the command line without an option the flag will be set to the NoOptDefVal. For example given:

```
var ip = flag.IntP("flagname", "f", 1234, "help message")
flag.Lookup("flagname").NoOptDefVal = "4321"
```

Would result in something like

Parsed Arguments	Resulting Value
--flagname=1357	ip=1357
--flagname	ip=4321
[nothing]	ip=1234

Command line flag syntax

```
--flag    // boolean flags, or flags with no option default values
--flag x  // only on flags without a default value
--flag=x
```

Unlike the flag package, a single dash before an option means something different than a double dash. Single dashes signify a series of shorthand letters for flags. All but the last shorthand letter must be boolean flags or a flag with a default value

```
// boolean or flags where the 'no option default value' is set
-f
-f=true
-abc
but
-b true is INVALID

// non-boolean and flags without a 'no option default value'
-n 1234
-n=1234
-n1234

// mixed
-abcs "hello"
-absd="hello"
-abcs1234
```

Flag parsing stops after the terminator "--". Unlike the flag package, flags can be interspersed with arguments anywhere on the command line before this terminator.

Integer flags accept 1234, 0664, 0x1234 and may be negative. Boolean flags (in their long form) accept 1, 0, t, f, true, false, TRUE, FALSE, True, False. Duration flags accept any input valid for time.ParseDuration.

Mutating or "Normalizing" Flag names

It is possible to set a custom flag name 'normalization function.' It allows flag names to be mutated both when created in the code and when used on the command line to some 'normalized' form. The 'normalized' form is used for comparison. Two examples of using the custom normalization func follow.

Example #1: You want -, _, and . in flags to compare the same. aka --my-flag == --my_flag == --my.flag

```
func wordSepNormalizeFunc(f *pflag.FlagSet, name string) pflag.NormalizedName {
    from := []string{"-", "_"}
    to := "."
    for _, sep := range from {
        name = strings.Replace(name, sep, to, -1)
    }
    return pflag.NormalizedName(name)
}

myFlagSet.SetNormalizeFunc(wordSepNormalizeFunc)
```

Example #2: You want to alias two flags. aka --old-flag-name == --new-flag-name

```
func aliasNormalizeFunc(f *pflag.FlagSet, name string) pflag.NormalizedName {
    switch name {
    case "old-flag-name":
        name = "new-flag-name"
        break
    }
    return pflag.NormalizedName(name)
}

myFlagSet.SetNormalizeFunc(aliasNormalizeFunc)
```

Deprecating a flag or its shorthand

It is possible to deprecate a flag, or just its shorthand. Deprecating a flag/shorthand hides it from help text and prints a usage message when the deprecated flag/shorthand is used.

Example #1: You want to deprecate a flag named "badflag" as well as inform the users what flag they should use instead.

```
// deprecate a flag by specifying its name and a usage message
flags.MarkDeprecated("badflag", "please use --good-flag instead")
```

This hides "badflag" from help text, and prints `Flag --badflag has been deprecated, please use --good-flag instead` when "badflag" is used.

Example #2: You want to keep a flag name "noshorthandflag" but deprecate its shortname "n".

```
// deprecate a flag shorthand by specifying its flag name and a usage message
flags.MarkShorthandDeprecated("noshorthandflag", "please use --noshorthandflag
only")
```

This hides the shortname "n" from help text, and prints `Flag shorthand -n has been deprecated, please use --noshorthandflag only` when the shorthand "n" is used.

Note that usage message is essential here, and it should not be empty.

Hidden flags

It is possible to mark a flag as hidden, meaning it will still function as normal, however will not show up in usage/help text.

Example: You have a flag named "secretFlag" that you need for internal use only and don't want it showing up in help text, or for its usage text to be available.

```
// hide a flag by specifying its name
flags.MarkHidden("secretFlag")
```

Disable sorting of flags

`pflag` allows you to disable sorting of flags for help and usage message.

Example:

```
flags.BoolP("verbose", "v", false, "verbose output")
flags.String("coolflag", "yeaah", "it's really cool flag")
flags.Int("usefulflag", 777, "sometimes it's very useful")
flags.SortFlags = false
flags.PrintDefaults()
```

Output:

```
  -v, --verbose           verbose output
      --coolflag string   it's really cool flag (default "yeaah")
      --usefulflag int    sometimes it's very useful (default 777)
```

Supporting Go flags when using pflag

In order to support flags defined using Go's `flag` package, they must be added to the `pflag` flagset. This is usually necessary to support flags defined by third-party dependencies (e.g. `golang/glog`).

Example: You want to add the Go flags to the `CommandLine` flagset

```
import (
    goflag "flag"
    flag "github.com/spf13/pflag"
)

var ip *int = flag.Int("flagname", 1234, "help message for flagname")

func main() {
    flag.CommandLine.AddGoFlagSet(goflag.CommandLine)
    flag.Parse()
}
```

More info

You can see the full reference documentation of the pflag package at godoc.org, or through go's standard documentation system by running `godoc -http=:6060` and browsing to http://localhost:6060/pkg/github.com/spf13/pflag after installation.