Glint Documentation

Release 0.1.0-beta

Mike Lowen

July 28, 2013

CONTENTS

Glint is a micro-framework for for providing command like functionality to a python command line application. Glint allows an application to accept command line arguments in the following fashion:

```
app.py <command> required_arg --optional_arg value --flag
```

CHAPTER

ONE

INSPIRATION

The inspiration for Glint came from wanting to have a command driven CLI appliation similar to how git works which I was unable to replicate with argparse.

CHAPTER

TWO

INSTALLATION

Glint requires Python 3.3 or higher to work, once you have downloaded the latest version from the release page at GitHub you need to run the following command from the base directory:

```
python setup.py install
```

At some point we will be submitting to the Python Package Index once that is done then it will be the preferred method to retreive and install Glint.

CHAPTER

THREE

QUICK START

The smallest working example is the example.py script below

```python
#! /bin/env python

import glint

def hello():
        print('Hello world.')

if __name__ == '__main__':
        runner = glint.Runner()
        runner['hello'] = hello
        runner.run()
```

This script provides you with two commands: the built in help command and the hello command which was defined in the script. The built in help command will output information that it has about the available commands that can be run, to view the help you would run the following command `./example.py help` which would produce output that looks like

```
usage: ./example.py <command>

Commands:

  help    Show this message and exit
  hello

See './example.py help --command <command>' for help on that command.
```

Running the hello command that is defined in the example looks like `./example.py hello` which produces the following expected output

```
Hello World.
```

For a more comprehensive look see the usage page.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

4.1 Commands

Commands are at the heart of Glint, a command defines what code should be run given certain user input. A command references a function and like a python function can take various arguments. In this section we are going to walk-through and build out the code from the quickstart to show how to define and use the different types of arguments a command can take.

4.1.1 Defining a Command

In the quick start we had a basic starting point

```
#! /bin/env python

import glint

def hello():
        print('Hello world.')

if __name__ == '__main__':
        runner = glint.Runner()
        runner['hello'] = hello
        runner.run()
```

The simplist way to define a command is once a `Runner` has been created to then assign a method to string

```
runner['hello'] = hello
```

Now when the command `./example.py hello` is run the hello function will be executed.

4.2 Arguments

Commands like the functions that they invoke can accept arguments, Glint supports three separate types of arguments: positional/required, optional and flags. This section walks you through adding one of each type of argument to our example script above.

4.2.1 Positional/Required arguments

Commands like the functions that they invoke can accept arguments, Glint supports three separate types of arguments the first of which is the position or required arguments. To define a positional argument you add a parameter to the function that the command invokes which has no default value, in our example it would look like the following.

```
def hello(name):
        print('Hello %s.' % name)
```

Now when we invoke the hello command we need to supply a value for name e.g. `./example.py hello mike` the string "mike" will be passed to the hello function in the name variable. When multiple positional arguments exist the values supplied will be assigned to the parameters in the order that the values are given. If the incorrect number of arguments are supplied then a error message will be printed out to the screen.

4.2.2 Optional arguments

The next argument that Glint supports is the optional argument, you define an optional argument by by adding a parameter to the function with a default value. In our example script we would update the hello function to look like the following.

```
def hello(name, message = None):
        print('Hello %s.' % name)

        if message is not None:
                print('I want to tell you: %s' % message)
```

To invoke an optional argument from the command line you need to supply the optional prefix (which by default is `--`) followed by the name of the parameter a space and then the value. For our example script it would look like.

```
./example.py hello mike --message "some stuff"
```

It doesn't matter where the optional argument is positioned they can be interspersed among the other arguments positional or otherwise.

4.2.3 Flags

The final type of argument that Glint supports in the flag. The flag is a boolean argument it is either true or false, to define a flag you add a parameter to your function which has the default value of `False` in our example script this would look like the following.

```
def hello(name, wave = False, message = None):
        print('Hello %s.' % name)

        if wave:
                print('I\'m waving at you!')

        if message is not None:
                print('I want to tell you: %s' % message)
```

To invoke a flag it is similar to an optional argument you supply the optional prefix followed by the name of the parameter, unlike the optional argument however there is no need to supply a value the presence of the flag is enough to tell Glint that the value should be set to true. For our example script invoking the flag would look like.

```
./example.py hello mike --wave
```

Also like the optional argument it doesn't matter where in the arguments the flag is supplied.

4.2.4 Final script

Now we've added the different types of arguments that Glint supports our complete example script looks like:

```
#! /bin/env python

import glint

def hello(name, wave = False, message = None):
        print('Hello %s.' % name)

        if wave:
                print('I\'m waving at you!')

        if message is not None:
                print('I want to tell you: %s' % message)

if __name__ == '__main__':
        runner = glint.Runner()
        runner['hello'] = hello
        runner.run()
```

4.3 Special Commands

Within Glint there are two special commands that work differently than what has previously been described. This section describes those special commands and how to work with them.

4.3.1 The None command

The `None` command is a special command for when no arguments are supplied to your application but rather than throwing an error you want a function to be run. To define the None command you assign a function in the runner with the command text of `None`, in our example script from above this would look like the following.

```
#! /bin/env python

import glint

def hello(name, wave = False, message = None):
        print('Hello %s.' % name)

        if wave:
                print('I\'m waving at you!')

        if message is not None:
                print('I want to tell you: %s' % message)

def default():
        print('No command has been supplied.')
```

```
if __name__ == '__main__':
        runner = glint.Runner()

        runner[None] = default
        runner['hello'] = hello

        runner.run()
```

Now when the script is run and no command is supplied then the message "No command has been supplied." will be printed to the console. It is important to note that currently the `None` command has the limitation that it cannot accept any arguments, if arguments are supplied then they will be ignored.

4.3.2 The help command

The help command is a command that has been built into Glint that prints out information regarding the available commands and their arguments to the console. This is invoked at the command line by running `./example.py help` to see the list of commands that are available, to see what parameters a particular command accepts you would run `./example.py help --command <command name>`. We go into more depth on the help command in the *help section*.

The help command

One of the built-in pieces of functionality that Glint provides is the help command which will display an automatically generated help screen for the user. This screen is generated by inspecting the commands which have been defined. In this section we will walk through building out the help data for the following example hello world application:

```
#! /bin/env python

import glint

def hello(name, wave = False, message = None):
        print('Hello %s.' % name)

        if wave:
                print('I\'m waving at you!')

        if message is not None:
                print('I want to tell you: %s' % message)

if __name__ == '__main__':
        runner = glint.Runner()
        runner['hello'] = (hello, 'Prints a hello message and exits.')
        runner.run()
```

When we run the built-in help command we get the following output:

```
usage: ./example.py <command>

Commands:

  help    Show this message and exit
  hello

See './example.py help --command <command>' for help on that command.
```

From that you see we can further inspect a command by calling `./example.py help --command hello` which will produce:

```
usage: ./example.py hello <name> [--message <message>] [--wave]

Arguments:

  name

Optional Arguments:

  --message

Flags:

  --wave
```

This gives us a good place to start but we can do better to bulk up what we’re telling the user. The first place we should start is give the user some context about what the call does, we can do this by supplying a description for the command. A command description is defined when a method is assigned, we change it from being assigned to just a method to a tuple containing the method and description like so

```
runner[’hello’] = (hello, ’Prints a hello message and exits.’)
```

Now when we run the help command we get.

```
usage: ./example.py <command>

Commands:

  hello   Prints a hello message and exits.
  help    Show this message and exit

See ’./example.py help --command <command>’ for help on that command.
```

That helps our users when trying to figure out what a command does, but it doesn’t do much for telling them what an argument means. To do that Glint hijacks the annotations of the parameters. If we are to do this in our example script we update the function signature to look like the following.

```
def hello(name: ’Who we are saying hello to.’, wave: ’Add this flag if you want to wave.’ = False, me
```

When we run the help command we won’t notice any change, though when the run help for the hello command

```
Prints a hello message and exits.

usage: ./example.py hello <name> [--message <message>] [--wave]

Arguments:

  name        Who we are saying hello to.

Optional Arguments:

  --message   A message we want to tell.

Flags:

  --wave      Add this flag if you want to wave.
```

We now have a lot more information that can help our users understand what our application does. The final script after we made the changes on this page ends up looking like.

```
#! /bin/env python

import glint

def hello(name: 'Who we are saying hello to.', wave: 'Add this flag if you want to wave.' = False, me
        print('Hello %s.' % name)

        if wave:
                print('I\'m waving at you!')

        if message is not None:
                print('I want to tell you: %s' % message)

if __name__ == '__main__':
        runner = glint.Runner()
        runner['hello'] = (hello, 'Prints a hello message and exits.')
        runner.run()
```

Inheritance

So far in all of our examples we've been creating a separate runner and assigning commands to it, on this page we are going to look at another way of setting up an application to use Glint via inheritance. Considering most of this content is covered else where in the documentation we're not going to step through the example, we're going to use the final script from the *commands section* and port it to use inheritance.

```
#! /bin/env python

import glint

class Application(glint.Runner):
        def __init__(self):
                glint.Runner.__init__(self)

                self[None] = self.default
                self['hello'] = self.hello

        def hello(self, name, wave = False, message = None):
                print('Hello %s.' % name)

                if wave:
                        print('I\'m waving at you!')

                if message is not None:
                        print('I want to tell you: %s' % message)

        def default(self):
                print('No command has been given to the application.')

if __name__ == '__main__':
        app = Application()
        app.run()
```

API - glint

`Runner([description = None[, show_usage = True[, prefix = '-`]]])]`

The Runner class is the core of the Glint framework it handles the routing of the commands to the correct methods.

Parameters

- **description** (*string or None*) – A string describing the application which will be printed out when the help command is run.
- **show_usage** (*bool*) – A flag to specify whether to add the help command.
- **prefix** (*string*) – The prefix used to identify optional arguments and flags.

`r[key] = value`

Assigns a method to be run when the command matches `key`, value can be one of two things:

- A function,
- A tuple containing a function and a description to be displayed when the help command is run.

`len(r)`

Returns the number of commands in the runner.

`key in r`

Returns if the current runner contains a command which matches `key`

`Runner.`**`run`**`(`[*args = None*]`)`

This method parses the arguments which are passed to it and runs the appropriate command. If the `args` parameter is `None` then `run` will use `sys.argv`.

Parameters **args** (*List or None*) – The arguments used to determine what command to run.

`Runner.`**`help`**`(`[*command = None*]`)`

This method returns a list of strings which contains help information. If the `show_usage` flag is true then this information will first be printed to stdout. If the `command` argument is not none then it will return help information specific to that command.

Parameters **command** (*string or None*) – The command to retrieve help information about.

Return type list of strings.

PYTHON MODULE INDEX