Roadmap Documentation

Release 0.1

Eric Martin

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Introduction and API

Roadmap is a routing library powered by regular expressions. Roadmap was created to quickly map large amounts of input to functions. In the particular case that sparked Roadmap's development, I was writing an IRC bot and I wanted a fast (to code and to execute) way to map input strings to functions. Beyond an IRC bot, I could also picture Roadmap being used to process data from a web API, user input, a socket, or really any stream of data.

Interface

Using Roadmap is hopefully very simple. The complete public interface of Roadmap consists of 3 methods.

`Router.`**`destination`**`(`*reg_str*, *pass_obj=True*`)`

Decorates functions to be called from a `roadmap.Router` instance.

Parameters

- **`reg_str`** – string that a regular expression can be created from
- **`pass_obj`** (*`boolean`*) – whether or not to pass object to decorated function

It is generally a good idea to use raw strings to create regular expressions. If the input string matches `reg_str`, the decorated function will be called.

`Router.`**`__init__`**`(`*processor*`)`

Creates a new `roadmap.Router` and assigns a processor.

Parameters **`processor`** – single variable function that *does something* with output

Because of Roadmap's coroutine based architechture, routing an object returns no value. At first it may seem strange that nothing can be returned, but the mindset of Roadmap is that you don't return values just for the sake of returning them; you return values to *do something* with them. In the case of my IRC bot, I wanted my functions to return strings. My `processor()` function received these strings and sent them over the socket. Other processing functions I can easily imagine are printing, logging, or commiting an object to a database.

This concept might be a little tricky, but the example will show how simple it actually is.

`Router.`**`route`**`(`*obj*, *key=None*`)`

Maps an object to its appropriate function

Parameters

- **`obj`** – object to route
- **`key`** – an optional string to route :obj‘obj‘ by

Return type `None`

`route()` is the method that will receive the input and begin the routing process for an object. The `key` parameter must be used if the object itself can’t be match by a regular expression, which, to the extent of my knowledge, means that that `obj` isn’t a string. Even if `obj` is matchable, `key` will be used if defined.

Details on Argument Passing

One slightly difficult aspect of using Roadmap is ensuring the desired arguments get passed to the routed functions. Briefly, here is the algorithm:

- The object routed (the primary input to *`roadmap.Router.route()`*) will be passed to the function unless `pass_obj` is set to `False` upon registering the function with the `Router` instance.
- If `pass_obj` is `True` and `obj` is iterable, each item in the iterable will be passed to the function. This feature may or may not be convenient, but it was useful for my IRC bot so I could nearly transparently route tuples.
- If unnamed groups are defined in the regular expression for the function, the value of each of these groups will be passed after the arguments from the object itself.
- If named groups are defined in the regular expression for the function, they will be passed as keyword arguments (`name=value`) to the function.

The examples should make this admittedly complex behavior a bit clearer.

CHAPTER 2

Examples

Let's create a basic `Router` instance to route email addresses. We will route them by subdomain. The goal is to respond to basic commands from string. We only want the bot to respond to commands, which will all be prefixed by `!`. The output of this Roadmap will just be printing. Therefore, it is logical for each handler to return a string. Here we go:

```
import roadmap.Router

def my_processor(string):
    print string

r = roadmap.Router(my_processor)

@r.destination(r'^echo (.*)', pass_obj=False)
def echo(message):
    return message

@r.destination(r'^divide ([0-9]+) ([0-9]+)', pass_obj=False)
def divide(dividend, divisor):
    return int(dividend) / float(divisor)

@r.destination(r'^test .*')
def test(string):
    return 'Input received: %s' % string

@r.destination(r'^intials (?P<last>),\W+(?P<first>)\W+(?P<middle>) ',\
                                                    pass_obj=False)
def initials(first, middle, last):
    return '%s. %s. %s.' % (first[0], middle[0], last[0])

for input in input_stream():
    if input.startswith('!'):
        r.route(input[1:])
```

Sorry this example is simple and extremely contrived, but it shows the main mechanics of Roadmap. Below, each part of the example is explained.

Initialization

```
def my_processor(string):
    print string

r = roadmap.Router(my_processor)
```

Each of the Router's function will be printable, and all we want to do with the output is print right now. The `my_processor()` function could just as easily handle instances as it handles strings in this function, and it save to database or communicate over a socket rather than just printing. `r` is created as a new `Router` that will use `my_processor()`.

Echo Function

```
@r.destination(r'echo (.*)', pass_obj=False)
def echo(message):
    return message
```

Any string that begins with `echo` will route to this function. `pass_obj` is set to `False`, so the full input string (including `echo`) will not be passed to the function, because there is no need for every message to being with `echo`. However, the regular expression does contain a group (indicated by the parenthesis) that will match any string. Unnamed groups are passed to the function in the order they are assigned. The regular expression group becomes `message` and is returned by the function, and then printed by `my_processor()`.

Add Function

```
@r.destination(r'^divide ([0-9]+) ([0-9]+)', pass_obj=False)
def divide(dividend, divisor):
    return int(dividend) / double(divisor)
```

`add()` follows the exact same rules as `echo()`, but simply shows that multiple unnamed groups can be used. The first group becomes `dividend`, the second group becomes `divisor`. Notice the type casting, because regular expression matches return strings.

Test Function

```
@r.destination(r'^test .*')
def test(string):
    return 'Input received: %s' % string
```

Note that no groups are captured in the regular expression. However, `pass_obj` is not specified, and defaults to `True`. Therefore, the string passed to *`roadmap.Router.route()`* where will be passed to `test()` as `string`.

Initials Function

```
@r.destination(r'^intials (?P<last>),\W+(?P<first>)\W+(?P<middle>) ',\
                                                    pass_obj=False)
def initials(first, middle, last):
    return '%s. %s. %s.' % (first[0], middle[0], last[0])
```

This is how named groups are handled by Roadmap. The order in the regular expression does not have to correspond to the order of the parameters.

Calling `route()`

```
for in_string in input_stream():
    if in_string.startswith('!'):
        r.route(in_string[1:])
```

This calls *`route()`* with the `!` stripped from the input string. Notice that *`roadmap.Router.route()`* returns no value.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index