Supybot Documentation

Release 0.83.4.1+git

Jeremiah Fincher and James McCoy

February 09, 2016

Contents

1 Advanced Plugin Config **3**
- 1.1 What's This Tutorial For? . . . 3
- 1.2 Using 'configure' effectively . . . 3
- 1.3 Using Config Groups . . . 5
- 1.4 Creating a Config Group . . . 5
- 1.5 Adding Values to a Group . . . 5
- 1.6 The Built-in Registry Types . . . 6
- 1.7 Custom Registry Types . . . 6

2 Advanced Plugin Testing **9**
- 2.1 Why Write Tests? . . . 9
- 2.2 Plugin Tests . . . 9
- 2.3 Plugin Test Methods . . . 11
- 2.4 Other Tests . . . 12

3 Capabilities **15**
- 3.1 Introduction . . . 15
- 3.2 User Capabilities . . . 15
- 3.3 Channel Capabilities . . . 16
- 3.4 Default Capabilities . . . 16
- 3.5 Final Word . . . 17

4 Configuration **19**
- 4.1 Introduction . . . 19
- 4.2 Configuration Registry . . . 19
- 4.3 Configuration Groups . . . 19
- 4.4 Configuration Values . . . 20
- 4.5 Default Values . . . 21
- 4.6 Searching the Registry . . . 21
- 4.7 Channel-Specific Configuration . . . 21
- 4.8 Editing the Configuration Values by Hand . . . 22

5 Frequently Asked Questions **23**

6 Getting Started with Supybot **27**
- 6.1 Introduction . . . 27
- 6.2 Initial Setup . . . 27
- 6.3 Listing Commands . . . 27
- 6.4 Making Supybot Recognize You . . . 28

6.5 Loading Plugins . . . 28
6.6 Getting More From Your Supybot . . . 29
6.7 Final Word . . . 29

7 Writing Your First Supybot Plugin **31**
7.1 Introduction . . . 31
7.2 README.txt . . . 32
7.3 __init__.py . . . 32
7.4 config.py . . . 33
7.5 plugin.py . . . 34
7.6 test.py . . . 37
7.7 Conclusion . . . 38

8 Style Guidelines **39**

9 Using Supybot's utils module **43**
9.1 str.py . . . 43
9.2 structures.py . . . 46
9.3 web.py . . . 47
9.4 The Best of the Rest . . . 48

10 Using commands.wrap to parse your command's arguments. **49**
10.1 Introduction . . . 49
10.2 Using Wrap . . . 49
10.3 Syntax Changes . . . 50
10.4 Customizing Wrap . . . 51
10.5 Converter List . . . 51
10.6 Contexts List . . . 55
10.7 Final Word . . . 55

11 Indices and tables **57**

Contents:

CHAPTER 1

Advanced Plugin Config

This tutorial covers some of the more advanced plugin config features available to Supybot plugin authors.

1.1 What's This Tutorial For?

Brief overview of what this tutorial covers and the target audience.

Want to know the crazy advanced features available to you, the Supybot plugin author? Well, this is the tutorial for you. This article assumes you've read the Supybot plugin author tutorial since all the basics of plugin config are handled there first.

In this tutorial we'll cover:

- Using the configure function more effectively by using the functions provided in supybot.questions
- Creating config variable groups and config variables underneath those groups.
- The built-in config variable types ("registry types") for use with config variables
- Creating custom registry types to handle config variable values more effectively

1.2 Using 'configure' effectively

How to use 'configure' effectively using the functions from 'supybot.questions'

In the original Supybot plugin author tutorial you'll note that we gloss over the configure portion of the config.py file for the sake of keeping the tutorial to a reasonable length. Well, now we're going to cover it in more detail.

The supybot.questions module is a nice little module coded specifically to help clean up the configure section of every plugin's config.py. The boilerplate config.py code imports the four most useful functions from that module:

- "expect" is a very general prompting mechanism which can specify certain inputs that it will accept and also specify a default response. It takes the following arguments:
  - prompt: The text to be displayed
  - possibilities: The list of possible responses (can be the empty list, [])
  - default (optional): Defaults to None. Specifies the default value to use if the user enters in no input.
  - acceptEmpty (optional): Defaults to False. Specifies whether or not to accept no input as an answer.
- "anything" is basically a special case of expect which takes anything (including no input) and has no default value specified. It takes only one argument:

  - prompt: The text to be displayed

- "something" is also a special case of expect, requiring some input and allowing an optional default. It takes the following arguments:
  - prompt: The text to be displayed
  - default (optional): Defaults to None. The default value to use if the user doesn't input anything.
- "yn" is for "yes or no" questions and basically forces the user to input a "y" for yes, or "n" for no. It takes the following arguments:
  - prompt: The text to be displayed
  - default (optional): Defaults to None. Default value to use if the user doesn't input anything.

All of these functions, with the exception of "yn", return whatever string results as the answer whether it be input from the user or specified as the default when the user inputs nothing. The "yn" function returns True for "yes" answers and False for "no" answers.

For the most part, the latter three should be sufficient, but we expose expect to anyone who needs a more specialized configuration.

Let's go through a quick example configure that covers all four of these functions. First I'll give you the code, and then we'll go through it, discussing each usage of a supybot.questions function just to make sure you realize what the code is actually doing. Here it is:

```
def configure(advanced): # This will be called by supybot to configure this module. advanced is # a bool
    that specifies whether the user identified himself as an advanced # user or not. You should effect
    your configuration by manipulating the # registry as appropriate. from supybot.questions import
    expect, anything, something, yn WorldDom = conf.registerPlugin('WorldDom', True) if yn("""The
    WorldDom plugin allows for total world domination

        with simple commands. Would you like these commands to be enabled for every-
        one?""", default=False):

      WorldDom.globalWorldDominationRequires.setValue("")

    else:

      cap = something("""What capability would you like to require for this command to be
        used?""", default="Admin")

      WorldDom.globalWorldDominationRequires.setValue(cap)

    dir = expect("""What direction would you like to attack from in

        your quest for world domination?""",

      ["north", "south", "east", "west", "ABOVE"], default="ABOVE")

    WorldDom.attackDirection.setValue(dir)
```

As you can see, this is the WorldDom plugin, which I am currently working on. The first thing our configure function checks is to see whether or not the bot owner would like the world domination commands in this plugin to be available to everyone. If they say yes, we set the globalWorldDominationRequires configuration variable to the empty string, signifying that no specific capabilities are necessary. If they say no, we prompt them for a specific capability to check for, defaulting to the "Admin" capability. Here they can create their own custom capability to grant to folks which this plugin will check for if they want, but luckily for the bot owner they don't really have to do this since Supybot's capabilities system can be flexed to take care of this.

Lastly, we check to find out what direction they want to attack from as they venture towards world domination. I prefer "death from above!", so I made that the default response, but the more boring cardinal directions are available as choices as well.

1.3 Using Config Groups

A brief overview of how to use config groups to organize config variables

Supybot's Hierarchical Configuration

Supybot's configuration is inherently hierarchical, as you've probably already figured out in your use of the bot. Naturally, it makes sense to allow plugin authors to create their own hierarchies to organize their configuration variables for plugins that have a lot of plugin options. If you've taken a look at the plugins that Supybot comes with, you've probably noticed that several of them take advantage of this. In this section of this tutorial we'll go over how to make your own config hierarchy for your plugin.

Here's the brilliant part about Supybot config values which makes hierarchical structuring all that much easier - values are groups. That is, any config value you may already defined in your plugins can already be treated as a group, you simply need to know how to add items to that group.

Now, if you want to just create a group that doesn't have an inherent value you can do that as well, but you'd be surprised at how rarely you have to do that. In fact if you look at most of the plugins that Supybot comes with, you'll only find that we do this in a handful of spots yet we use the "values as groups" feature quite a bit.

1.4 Creating a Config Group

As stated before, config variables themselves are groups, so you can create a group simply by creating a configuration variable:

conf.registerGlobalValue(WorldDom, 'globalWorldDominationRequires',

registry.String('', """Determines the capability required to access the world domination commands in this plugin."""))

As you probably know by now this creates the config variable supybot.plugins.WorldDom.globalWorldDominationRequires which you can access/set using the Config plugin directly on the running bot. What you may not have known prior to this tutorial is that that variable is also a group. Specifically, it is now the WorldDom.globalWorldDominationRequires group, and we can add config variables to it! Unfortunately, this particular bit of configuration doesn't really require anything underneath it, so let's create a new group which does using the "create only a group, not a value" command.

Let's create a configurable list of targets for different types of attacks (land, sea, air, etc.). We'll call the group attackTargets. Here's how you create just a config group alone with no value assigned:

conf.registerGroup(WorldDom, 'attackTargets')

The first argument is just the group under which you want to create your new group (and we got WorldDom from conf.registerPlugin which was in our boilerplate code from the plugin creation wizard). The second argument is, of course, the group name. So now we have WorldDom.attackTargets (or, fully, supybot.plugins.WorldDom.attackTargets).

1.5 Adding Values to a Group

Actually, you've already done this several times, just never to a custom group of your own. You've always added config values to your plugin's config group. With that in mind, the only slight modification needed is to simply point to the new group:

conf.registerGlobalValue(WorldDom.attackTargets, 'air',

registry.SpaceSeparatedListOfStrings('', """Contains the list of air targets."""))

And now we have a nice list of air targets! You'll notice that the first argument is WorldDom.attackTargets, our new group. Make sure that the conf.registerGroup call is made before this one or else you'll get a nasty AttributeError.

1.6 The Built-in Registry Types

A rundown of all of the built-in registry types available for use with config variables.

The "registry" module defines the following config variable types for your use (I'll include the 'registry.' on each one since that's how you'll refer to it in code most often). Most of them are fairly self-explanatory, so excuse the boring descriptions:

- **registry.Boolean - A simple true or false value. Also accepts the** following for true: "true", "on" "enable", "enabled", "1", and the following for false: "false", "off", "disable", "disabled", "0",
- registry.Integer - Accepts any integer value, positive or negative.
- registry.NonNegativeInteger - Will hold any non-negative integer value.
- **registry.PositiveInteger - Same as above, except that it doesn't accept 0** as a value.
- registry.Float - Accepts any floating point number.
- registry.PositiveFloat - Accepts any positive floating point number.
- **registry.Probability - Accepts any floating point number between 0 and 1** (inclusive, meaning 0 and 1 are also valid).
- registry.String - Accepts any string that is not a valid Python command
- **registry.NormalizedString - Accepts any string (with the same exception** above) but will normalize sequential whitespace to a single space..
- **registry.StringSurroundedBySpaces - Accepts any string but assures that** it has a space preceding and following it. Useful for configuring a string that goes in the middle of a response.
- **registry.StringWithSpaceOnRight - Also accepts any string but assures** that it has a space after it. Useful for configuring a string that begins a response.
- registry.Regexp - Accepts only valid (Perl or Python) regular expressions
- **registry.SpaceSeparatedListOfStrings - Accepts a space-separated list of** strings.

There are a few other built-in registry types that are available but are not usable in their current state, only by creating custom registry types, which we'll go over in the next section.

1.7 Custom Registry Types

How to create and use your own custom registry types for use in customizing plugin config variables.

Why Create Custom Registry Types?

For most configuration, the provided types in the registry module are sufficient. However, for some configuration variables it's not only convenient to use custom registry types, it's actually recommended. Customizing registry types allows for tighter restrictions on the values that get set and for greater error-checking than is possible with the provided types.

What Defines a Registry Type?

First and foremost, it needs to subclass one of the existing registry types from the registry module, whether it be one of the ones in the previous section or one of the other classes in registry specifically designed to be subclassed.

Also it defines a number of other nice things: a custom error message for your type, customized value-setting (transforming the data you get into something else if wanted), etc.

Creating Your First Custom Registry Type

As stated above, priority number one is that you subclass one of the types in the registry module. Basically, you just subclass one of those and then customize whatever you want. Then you can use it all you want in your own plugins. We'll do a quick example to demonstrate.

We already have registry.Integer and registry.PositiveInteger, but let's say we want to accept only negative integers. We can create our own NegativeInteger registry type like so:

```
class NegativeInteger(registry.Integer): """Value must be a negative integer."""
    def setValue(self, v):
        if v >= 0: self.error()
        registry.Integer.setValue(self, v)
```

All we need to do is define a new error message for our custom registry type (specified by the docstring for the class), and customize the setValue function. Note that all you have to do when you want to signify that you've gotten an invalid value is to call self.error(). Finally, we call the parent class's setValue to actually set the value.

What Else Can I Customize?

Well, the error string and the setValue function are the most useful things that are available for customization, but there are other things. For examples, look at the actual built-in registry types defined in registry.py (in the src directory distributed with the bot).

What Subclasses Can I Use?

Chances are one of the built-in types in the previous section will be sufficient, but there are a few others of note which deserve mention:

- **registry.Value - Provides all the core functionality of registry types** (including acting as a group for other config variables to reside underneath), but nothing more.
- **registry.OnlySomeStrings - Allows you to specify only a certain set of** strings as valid values. Simply override validStrings in the inheriting class and you're ready to go.
- **registry.SeparatedListOf - The generic class which is the parent class to** registry.SpaceSeparatedListOfStrings. Allows you to customize four things: the type of sequence it is (list, set, tuple, etc.), what each item must be (String, Boolean, etc.), what separates each item in the sequence (using custom splitter/joiner functions), and whether or not the sequence is to be sorted. Look at the definitions of registry.SpaceSeparatedListOfStrings and registry.CommaSeparatedListOfStrings at the bottom of registry.py for more information. Also, there will be an example using this in the section below.

Using My Custom Registry Type

Using your new registry type is relatively straightforward. Instead of using whatever registry built-in you might have used before, now use your own custom class. Let's say we define a registry type to handle a comma-separated list of probabilities:

```
class CommaSeparatedListOfProbabilities(registry.SeparatedListOf): Value = registry.Probability
    def splitter(self, s):
        return re.split(r's*,s*', s)
    joiner = ', '.join
```

Now, to use that type we simply have to specify it whenever we create a config variable using it:

```
conf.registerGlobalValue(SomePlugin, 'someConfVar', CommaSeparatedListOfProbabilities('0.0,
    1.0', """Holds the list of probabilities for whatever."""))
```

Note that we initialize it just the same as we do any other registry type, with two arguments: the default value, and then the description of the config variable.

CHAPTER 2

Advanced Plugin Testing

The complete guide to writing tests for your plugins.

2.1 Why Write Tests?

Why should I write tests for my plugin? Here's why.

For those of you asking "Why should I write tests for my plugin? I tried it out, and it works!", read on. For those of you who already realize that Testing is Good (TM), skip to the next section.

Here are a few quick reasons why to test your Supybot plugins.

- When/if we rewrite or change certain features in Supybot, tests make

sure your plugin will work with these changes. It's much easier to run supybot-test MyPlugin after upgrading the code and before even reloading the bot with the new code than it is to load the bot with new code and then load the plugin only to realize certain things don't work. You may even ultimately decide you want to stick with an older version for a while as you patch your custom plugin. This way you don't have to rush a patch while restless users complain since you're now using a newer version that doesn't have the plugin they really like.

- Running the automated tests takes a few seconds, testing plugins in IRC

on a live bot generally takes quite a bit longer. We make it so that writing tests generally doesn't take much time, so a small initial investment adds up to lots of long-term gains.

- If you want your plugin to be included in any of our releases (the core

Supybot if you think it's worthy, or our supybot-plugins package), it has to have tests. Period.

For a bigger list of why to write unit tests, check out this article:

http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html

and also check out what the Extreme Programming folks have to say about unit tests:

http://www.extremeprogramming.org/rules/unittests.html

2.2 Plugin Tests

How to write tests for commands in your plugins.

Introduction

This tutorial assumes you've read through the plugin author tutorial, and that you used supybot-plugin-create to create your plugin (as everyone should). So, you should already have all the necessary imports and all that boilerplate stuff in test.py already, and you have already seen what a basic plugin test looks like from the plugin author tutorial. Now we'll go into more depth about what plugin tests are available to Supybot plugin authors.

Plugin Test Case Classes

Supybot comes with two plugin test case classes, PluginTestCase and ChannelPluginTestCase. The former is used when it doesn't matter whether or not the commands are issued in a channel, and the latter is used for when it does. For the most part their API is the same, so unless there's a distinction between the two we'll treat them as one and the same when discussing their functionality.

The Most Basic Plugin Test Case

At the most basic level, a plugin test case requires three things:

- the class declaration (subclassing PluginTestCase or ChannelPluginTestCase)
- a list of plugins that need to be loaded for these tests (does not include Owner, Misc, or Config, those are always automatically loaded) - often this is just the name of the plugin that you are writing tests for
- some test methods

Here's what the most basic plugin test case class looks like (for a plugin named MyPlugin):

```
class MyPluginTestCase(PluginTestCase): plugins = ('MyPlugin',)

    def testSomething(self): # assertions and such go here
```

Your plugin test case should be named TestCase as you see above, though it doesn't necessarily have to be named that way (supybot-plugin-create puts that in place for you anyway). As you can see we elected to subclass PluginTestCase because this hypothetical plugin apparently doesn't do anything channel-specific.

As you probably noticed, the plugins attribute of the class is where the list of necessary plugins goes, and in this case just contains the plugin that we are testing. This will be the case for probably the majority of plugins. A lot of the time test writers will use a bot function that performs some function that they don't want to write code for and they will just use command nesting to feed the bot what they need by using that plugin's functionality. If you choose to do this, only do so with core bot plugins as this makes distribution of your plugin simpler. After all, we want people to be able to run your plugin tests without having to have all of your plugins!

One last thing to note before moving along is that each of the test methods should describe what they are testing. If you want to test that your plugin only responds to registered users, don't be afraid to name your test method testOnlyRespondingToRegisteredUsers or testNotRespondingToUnregisteredUsers. You may have noticed some rather long and seemingly unwieldy test method names in our code, but that's okay because they help us know exactly what's failing when we run our tests. With an ambiguously named test method we may have to crack open test.py after running the tests just to see what it is that failed. For this reason you should also test only one thing per test method. Don't write a test method named testFoobarAndBaz. Just write two test methods, testFoobar and testBaz. Also, it is important to note that test methods must begin with test and that any method within the class that does begin with test will be run as a test by the supybot-test program. If you want to write utility functions in your test class that's fine, but don't name them something that begins with test or they will be executed as tests.

Including Extra Setup

Some tests you write may require a little bit of setup. For the most part it's okay just to include that in the individual test method itself, but if you're duplicating a lot of setup code across all or most of your test methods it's best to use the setUp method to perform whatever needs to be done prior to each test method.

The setUp method is inherited from the whichever plugin test case class you chose for your tests, and you can add whatever functionality you want to it. Note the important distinction, however: you should be adding to it and not

overriding it. Just define setUp in your own plugin test case class and it will be run before all the test methods are invoked.

Let's do a quick example of one. Let's write a setUp method which registers a test user for our test bot:

def setUp(self): ChannelPluginTestCase.setUp(self) # important!! # Create a valid user to use self.prefix = 'foo!bar@baz' self.feedMsg('register tester moo', to=self.nick, frm=self.prefix)) m = self.getMsg() # Response to registration.

Now notice how the first line calls the parent class's setUp method first? This must be done first. Otherwise several problems are likely to arise. For one, you wouldn't have an irc object at self.irc that we use later on nor would self.nick be set.

As for the rest of the method, you'll notice a few things that are available to the plugin test author. self.prefix refers to the hostmask of the hypothetical test user which will be "talking" to the bot, issuing commands. We set it to some generically fake hostmask, and then we use feedMsg to send a private message (using the bot's nick, accessible via self.nick) to the bot registering the username "tester" with the password "moo". We have to do it this way (rather than what you'll find out is the standard way of issuing commands to the bot in test cases a little later) because registration must be done in private. And lastly, since feedMsg doesn't dequeue any messages from the bot after being fed a message, we perform a getMsg to get the response. You're not expected to know all this yet, but do take note of it since using these methods in test-writing is not uncommon. These utility methods as well as all of the available assertions are covered in the next section.

So, now in any of the test methods we write, we'll be able to count on the fact that there will be a registered user "tester" with a password of "moo", and since we changed our prefix by altering self.prefix and registered after doing so, we are now identified as this user for all messages we send unless we specify that they are coming from some other prefix.

The Opposite of Setting-up: Tearing Down

If you did some things in your setUp that you want to clean up after, then this code belongs in the tearDown method of your test case class. It's essentially the same as setUp except that you probably want to wait to invoke the parent class's tearDown until after you've done all of your tearing down. But do note that you do still have to invoke the parent class's tearDown method if you decide to add in your own tear-down stuff.

Setting Config Variables for Testing

Before we delve into all of the fun assertions we can use in our test methods it's worth noting that each plugin test case can set custom values for any Supybot config variable they want rather easily. Much like how we can simply list the plugins we want loaded for our tests in the plugins attribute of our test case class, we can set config variables by creating a mapping of variables to values with the config attribute.

So if, for example, we wanted to disable nested commands within our plugin testing for some reason, we could just do this:

class MyPluginTestCase(PluginTestCase): config = {'supybot.commands.nested': False}

def testThisThing(self): # stuff

And now you can be assured that supybot.commands.nested is going to be off for all of your test methods in this test case class.

2.3 Plugin Test Methods

The full list of test methods and how to use them.

Introduction

You know how to make plugin test case classes and you know how to do just about everything with them except to actually test stuff. Well, listed below are all of the assertions used in tests. If you're unfamiliar with what an assertion

is in code testing, it is basically a requirement of something that must be true in order for that test to pass. It's a necessary condition. If any assertion within a test method fails the entire test method fails and it goes on to the next one.

Assertions

All of these are methods of the plugin test classes themselves and hence are accessed by using self.assertWhatever in your test methods. These are sorted in order of relative usefulness.

- **assertResponse(query, expectedResponse) - Feeds query to the bot as a** message and checks to make sure the response is expectedResponse. The test fails if they do not match (note that prefixed nicks in the response do not need to be included in the expectedResponse).
- **assertError(query) - Feeds query to the bot and expects an error in** return. Fails if the bot doesn't return an error.
- **assertNotError(query) - The opposite of assertError. It doesn't matter** what the response to query is, as long as it isn't an error. If it is not an error, this test passes, otherwise it fails.
- **assertRegexp(query, regexp, flags=re.I) - Feeds query to the bot and** expects something matching the regexp (no m// required) in regexp with the supplied flags. Fails if the regexp does not match the bot's response.
- **assertNotRegexp(query, regexp, flags=re.I) - The opposite of** assertRegexp. Fails if the bot's output matches regexp with the supplied flags.
- **assertHelp(query) - Expects query to return the help for that command.** Fails if the command help is not triggered.
- **assertAction(query, expectedResponse=None) - Feeds query to the bot and** expects an action in response, specifically expectedResponse if it is supplied. Otherwise, the test passes for any action response.
- **assertActionRegexp(query, regexp, flags=re.I) - Basically like** assertRegexp but carries the extra requirement that the response must be an action or the test will fail.

Utilities

- **feedMsg(query, to=None, frm=None) - Simply feeds query to whoever is** specified in to or to the bot itself if no one is specified. Can also optionally specify the hostmask of the sender with the frm keyword. Does not actually perform any assertions.
- getMsg(query) - Feeds query to the bot and gets the response.

2.4 Other Tests

If you had to write helper code for a plugin and want to test it, here's how.

Previously we've only discussed how to test stuff in the plugin that is intended for IRC. Well, we realize that some Supybot plugins will require utility code that doesn't necessarily require all of the overhead of setting up IRC stuff, and so we provide a more lightweight test case class, SupyTestCase, which is a very very light wrapper around unittest.TestCase (from the standard unittest module) that basically just provides a little extra logging. This test case class is what you should use for writing those test cases which test things that are independent of IRC.

For example, in the MoobotFactoids plugin there is a large chunk of utility code dedicating to parsing out random choices within a factoid using a class called OptionList. So, we wrote the OptionListTestCase as a SupyTestCase for the MoobotFactoids plugin. The setup for test methods is basically the same as before, only you don't have to define plugins since this is independent of IRC.

You still have the choice of using setUp and tearDown if you wish, since those are inherited from unittest.TestCase. But, the same rules about calling the setUp or tearDown method from the parent class still apply.

With all this in hand, now you can write great tests for your Supybot plugins!

CHAPTER 3

Capabilities

3.1 Introduction

Ok, some explanation of the capabilities system is probably in order. With most IRC bots (including the ones I've written myself prior to this one) "what a user can do" is set in one of two ways. On the *really* simple bots, each user has a numeric "level" and commands check to see if a user has a "high enough level" to perform some operation. On bots that are slightly more complicated, users have a list of "flags" whose meanings are hardcoded, and the bot checks to see if a user possesses the necessary flag before performing some operation. Both methods, IMO, are rather arbitrary, and force the user and the programmer to be unduly confined to less expressive constructs.

This bot is different. Every user has a set of "capabilities" that is consulted every time they give the bot a command. Commands, rather than checking for a user level of 100, or checking if the user has an 'o' flag, are instead able to check if a user has the 'owner' capability. At this point such a difference might not seem revolutionary, but at least we can already tell that this method is self-documenting, and easier for users and developers to understand what's truly going on.

3.2 User Capabilities

What the heck can these capabilities DO?

If that was all, well, the capability system would be *cool*, but not many people would say it was *awesome*. But it **is** awesome! Several things are happening behind the scenes that make it awesome, and these are things that couldn't happen if the bot was using numeric userlevels or single-character flags. First, whenever a user issues the bot a command, the command dispatcher checks to make sure the user doesn't have the "anticapability" for that command. An anticapability is a capability that, instead of saying "what a user can do", says what a user *cannot* do. It's formed rather simply by adding a dash ('-') to the beginning of a capability; 'rot13' is a capability, and '-rot13' is an anticapability.

Anyway, when a user issues the bot a command, perhaps 'calc' or 'help', the bot first checks to make sure the user doesn't have the '-calc' or the '-help' (anti)capabilities before even considering responding to the user. So commands can be turned on or off on a *per user* basis, offering fine-grained control not often (if at all!) seen in other bots. This can be further refined by limiting the (anti)capability to a command in a specific plugin or even an entire plugin. For example, the rot13 command is in the Filter plugin. If a user should be able to use another rot13 command, but not the one in the Format plugin, they would simply need to be given '-Format.rot13' anticapability. Similarly, if a user were to be banned from using the Filter plugin altogether, they would simply need to be given the '-Filter' anticapability.

3.3 Channel Capabilities

What if #linux wants completely different capabilities from #windows?

But that's not all! The capabilities system also supports *channel* capabilities, which are capabilities that only apply to a specific channel; they're of the form '#channel,capability'. Whenever a user issues a command to the bot in a channel, the command dispatcher also checks to make sure the user doesn't have the anticapability for that command *in that channel*, and if the user does, the bot won't respond to the user in the channel. Thus now, in addition to having the ability to turn individual commands on or off for an individual user, we can now turn commands on or off for an individual user on an individual channel!

So when a user 'foo' sends a command 'bar' to the bot on channel '#baz', first the bot checks to see if the user has the anticapability for the command by itself, '-bar'. If so, it errors right then and there, telling the user that he lacks the 'bar' capability. If the user doesn't have that anticapability, then the bot checks to see if the user issued the command over a channel, and if so, checks to see if the user has the antichannelcapability for that command, '#baz,-bar'. If so, again, he tells the user that he lacks the 'bar' capability. If neither of these anticapabilities are present, then the bot just responds to the user like normal.

3.4 Default Capabilities

So what capabilities am I dealing with already?

There are several default capabilities the bot uses. The most important of these is the 'owner' capability. This capability allows the person having it to use *any* command. It's best to keep this capability reserved to people who actually have access to the shell the bot is running on. It's so important, in fact, that the bot will not allow you to add it with a command–you'll have you edit the users file directly to give it to someone.

There is also the 'admin' capability for non-owners that are highly trusted to administer the bot appropriately. They can do things such as change the bot's nick, cause the bot to ignore a given user, make the bot join or part channels, etc. They generally cannot do administration related to channels, which is reserved for people with the next capability.

People who are to administer channels with the bot should have the '#channel,op' capability–whatever channel they are to administrate, they should have that channel capability for 'op'. For example, since I want inkedmn to be an administrator in #supybot, I'll give him the '#supybot,op' capability. This is in addition to his 'admin' capability, since the 'admin' capability doesn't give the person having it control over channels. '#channel,op' is used for such things as giving/receiving ops, kickbanning people, lobotomizing the bot, ignoring users in the channel, and managing the channel capabilities. The '#channel,op' capability is also basically the equivalent of the 'owner' capability for capabilities involving #channel–basically anyone with the #channel,op capability is considered to have all positive capabilities and no negative capabilities for #channel.

One other globally important capability exists: 'trusted'. This is a command that basically says "This user can be trusted not to try and crash the bot." It allows users to call commands like 'icalc' in the 'Math' plugin, which can cause the bot to begin a calculation that could potentially never return (a calculation like '10**10**10**10'). Another command that requires the 'trusted' capability is the 're' command in the 'Utilities' plugin, which (due to the regular expression implementation in Python (and any other language that uses NFA regular expressions, like Perl or Ruby or Lua or ...) which can allow a regular expression to take exponential time to process). Consider what would happen if someone gave the bot the command 're [format join "" s/./ [dict go] /] [dict go]' It would basically replace every character in the output of 'dict go' (14,896 characters!) with the entire output of 'dict go', resulting in 221MB of memory allocated! And that's not even the worst example!

3.5 Final Word

From a programmer's perspective, capabilties are flexible and easy to use. Any command can check if a user has any capability, even ones not thought of when the bot was originally written. Plugins can easily add their own capabilities– it's as easy as just checking for a capability and documenting somewhere that a user needs that capability to do something.

From an user's perspective, capabilities remove a lot of the mystery and esotery of bot control, in addition to giving a bot owner absolutely finegrained control over what users are allowed to do with the bot. Additionally, defaults can be set by the bot owner for both individual channels and for the bot as a whole, letting an end-user set the policy he wants the bot to follow for users that haven't yet registered in his user database. It's really a revolution!

CHAPTER 4

Configuration

4.1 Introduction

So you've got your Supybot up and running and there are some things you don't like about it. Fortunately for you, chances are that these things are configurable, and this document is here to tell you how to configure them.

Configuration of Supybot is handled via the *Config* plugin, which controls runtime access to Supybot's registry (the configuration file generated by the 'supybot-wizard' program you ran). The *Config* plugin provides a way to get or set variables, to list the available variables, and even to get help for certain variables. Take a moment now to read the help for each of those commands: `config`, `list`, and `help`. If you don't know how to get help on those commands, take a look at the GETTING_STARTED document.

4.2 Configuration Registry

Now, if you're used to the Windows registry, don't worry, Supybot's registry is completely different. For one, it's completely plain text. There's no binary database sensitive to corruption, it's not necessary to use another program to edit it–all you need is a simple text editor. But there is at least one good idea in Windows' registry: hierarchical configuration.

Supybot's configuration variables are organized in a hierarchy: variables having to do with the way Supybot makes replies all start with *supybot.reply*; variables having to do with the way a plugin works all start with *supybot.plugins.Plugin* (where 'Plugin' is the name of the plugin in question). This hierarchy is nice because it means the user isn't inundated with hundreds of unrelated and unsorted configuration variables.

Some of the more important configuration values are located directly under the base group, *supybot*. Things like the bot's nick, its ident, etc. Along with these config values are a few subgroups that contain other values. Some of the more prominent subgroups are: *plugins* (where all the plugin-specific configuration is held), *reply* (where variables affecting the way a Supybot makes its replies resides), *replies* (where all the specific standard replies are kept), and *directories* (where all the directories a Supybot uses are defined). There are other subgroups as well, but these are the ones we'll use in our example.

4.3 Configuration Groups

Using the *Config* plugin, you can list values in a subgroup and get or set any of the values anywhere in the configuration hierarchy. For example, let's say you wanted to see what configuration values were under the *supybot* (the base group) hierarchy. You would simply issue this command:

```
<jemfinch|lambda> @config list supybot
<supybot> jemfinch|lambda: @abuse, @capabilities, @commands,
          @databases, @debug, @directories, @drivers, @log, @networks,
          @nick, @plugins, @protocols, @replies, @reply,
          alwaysJoinOnInvite, channels, defaultIgnore,
          defaultSocketTimeout, externalIP, flush,
          followIdentificationThroughNickChanges, ident, pidFile,
          snarfThrottle, upkeepInterval, and user
```

These are all the configuration groups and values which are under the base *supybot* group. Actually, their full names would each have a 'supybot.' prepended to them, but it is omitted in the listing in order to shorten the output. The first entries in the output are the groups (distinguished by the '@' symbol in front of them), and the rest are the configuration values. The '@' symbol (like the '#' symbol we'll discuss later) is simply a visual cue and is not actually part of the name.

4.4 Configuration Values

Okay, now that you've used the Config plugin to list configuration variables, it's time that we start looking at individual variables and their values.

The first (and perhaps most important) thing you should know about each configuration variable is that they all have an associated help string to tell you what they represent. So the first command we'll cover is `config help`. To see the help string for any value or group, simply use the `config help` command. For example, to see what this *supybot.snarfThrottle* configuration variable is all about, we'd do this:

```
<jemfinch|lambda> @config help supybot.snarfThrottle
<supybot> jemfinch|lambda: A floating point number of seconds to
          throttle snarfed URLs, in order to prevent loops between two
          bots snarfing the same URLs and having the snarfed URL in
          the output of the snarf message.  (Current value: 10.0)
```

Pretty simple, eh?

Now if you're curious what the current value of a configuration variable is, you'll use the `config` command with one argument, the name of the variable you want to see the value of:

```
<jemfinch|lambda> @config supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: '@'
```

To set this value, just stick an extra argument after the name:

```
<jemfinch|lambda> @config supybot.reply.whenAddressedBy.chars @$
<supybot> jemfinch|lambda: The operation succeeded.
```

Now check this out:

```
<jemfinch|lambda> $config supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: '@$'
```

Note that we used '$' as our prefix character, and that the value of the configuration variable changed. If I were to use the `flush` command now, this change would be flushed to the registry file on disk (this would also happen if I made the bot quit, or pressed Ctrl-C in the terminal which the bot was running). Instead, I'll revert the change:

```
<jemfinch|lambda> $config supybot.reply.whenAddressedBy.chars @
<supybot> jemfinch|lambda: The operation succeeded.
<jemfinch|lambda> $note that this makes no response.
```

4.5 Default Values

If you're ever curious what the default for a given configuration variable is, use the `config default` command:

```
<jemfinch|lambda> @config default supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: ''
```

Thus, to reset a configuration variable to its default value, you can simply say:

```
<jemfinch|lambda> @config supybot.reply.whenAddressedBy.chars [config
                  default supybot.reply.whenAddressedBy.chars]
<supybot> jemfinch|lambda: The operation succeeded.
<jemfinch|lambda> @note that this does nothing
```

Simple, eh?

4.6 Searching the Registry

Now, let's say you want to find all configuration variables that might be even remotely related to opping. For that, you'll want the `config search` command. Check this out:

```
<jemfinch|lamda> @config search op
<supybot> jemfinch|lambda: supybot.plugins.Enforcer.autoOp,
          supybot.plugins.Enforcer.autoHalfop,
          supybot.plugins.Enforcer.takeRevenge.onOps,
          supybot.plugins.Enforcer.cycleToGetOps,
          supybot.plugins.Topic, supybot.plugins.Topic.public,
          supybot.plugins.Topic.separator,
          supybot.plugins.Topic.format,
          supybot.plugins.Topic.recognizeTopiclen,
          supybot.plugins.Topic.default,
          supybot.plugins.Topic.undo.max,
          supybot.plugins.Relay.topicSync
```

Sure, it showed all the topic-related stuff in there, but it also showed you all the op-related stuff, too. Do note, however, that you can only see configuration variables for plugins that are currently loaded or that you loaded in the past; if you've never loaded a plugin there's no way for the bot to know what configuration variables it registers.

4.7 Channel-Specific Configuration

Many configuration variables can be specific to individual channels. The *Config* plugin provides an easy way to configure something for a specific channel; for instance, in order to set the prefix chars for a specific channel, do this in that channel:

```
<jemfinch|lambda> @config channel supybot.reply.whenAddressedBy.chars !
<supybot> jemfinch|lambda: The operation succeeded.
```

That'll set the prefix chars in the channel from which the message was sent to '!'. Voila, channel-specific values! Also, note that when using the *Config* plugin's `list` command, channel-specific values are preceeded by a '#' character to indicate such (similar to how '@' is used to indicate a group of values).

4.8 Editing the Configuration Values by Hand

Some people might like editing their registry file directly rather than manipulating all these things through the bot. For those people, we offer the `config reload` command, which reloads both registry configuration and user/channel/ignore database configuration.

Just edit the interesting files and then give the bot the `config reload` command and it'll work as expected. Do note, however, that Supybot flushes his configuration files and database to disk every hour or so, and if this happens after you've edited your configuration files but before you reload your changes, you could lose the changes you made. To prevent this, set the *supybot.flush* value to 'Off' while editing the files, and no automatic flushing will occur.

CHAPTER 5

Frequently Asked Questions

How do I make my Supybot connect to multiple servers?

Just use the *connect* command in the *Network* plugin.

Why does my bot not recognize me or tell me that I don't have the 'owner' capability?

Because you've not given it anything to recognize you from!

You'll need to identify with the bot (`help identify` to see how that works) or add your hostmask to your user record (`help hostmask add` to see how that works) for it to know that you're you.

You may wish to note that addhostmask can accept a password; rather than identify, you can send the command:

```
hostmask add myOwnerUser [hostmask] myOwnerUserPassword
```

and the bot will add your current hostmask to your owner user (of course, you should change myOwnerUser and myOwnerUserPassword appropriately for your bot).

What is a hostmask?

Each user on IRC is uniquely identified by a string which we call a *hostmask*. The IRC RFC refers to it as a prefix. Either way, it consists of a nick, a user, and a host, in the form `nick!user@host`. If your Supybot complains that something you've given to it isn't a hostmask, make sure that you have those three components and that they're joined in the appropriate manner.

My bot can't handle nicks with brackets in them!

It always complains about something not being a valid command, or about spurious or missing right brackets, etc.

You should quote arguments (using double quotes, like this: `"foo[bar]"`) that have brackets in them that you don't wish to be evaluated as nested commands. Otherwise, you can turn off nested commands by setting *supybot.commands.nested* to False, or change the brackets that nest commands, by setting *supybot.commands.nested.brackets* to some other value (like <>, which can't occur in IRC nicks).

I added an alias, but it doesn't work!

Take a look at `help <alias you added>`. If the alias the bot has listed doesn't match what you're giving it, chances are you need to quote your alias in order for the brackets not to be evaluated. For instance, if you're adding an alias to give you a link to your homepage, you need to say:

```
alias add mylink "format concat http://my.host.com/ [urlquote $1]"
```

and not:

```
alias add mylink format concat http://my.host.com/ [urlquote $1]
```

The first version works; the second version will always return the same url.

What does 'lobotomized' mean?

I see this word in commands and in my *channels.conf*, but I don't know what it means. What does Supybot mean when it says "lobotomized"?

A lobotomy is an operation that removes the frontal lobe of the brain, the part that does most of a person's thinking. To "lobotomize" a bot is to tell it to stop thinking–thus, a lobotomized bot will not respond to anything said by anyone other than its owner in whichever channels it is lobotomized.

The term is certainly suboptimal, but remains in use because it was historically used by certain other IRC bots, and we wanted to ease the transition to Supybot from those bots by reusing as much terminology as possible.

Is there a way to load all the plugins Supybot has?

No, there isn't. Even if there were, some plugins conflict with other plugins, so it wouldn't make much sense to load them. For instance, what would a bot do with *Factoids*, *MoobotFactoids*, and *Infobot* all loaded? Probably just annoy people :)

If you want to know more about the plugins that are available, check out our plugin index at our website.

Is there a command that can tell me what capability another command requires?

No, there isn't, and there probably never will be.

Commands have the flexibility to check any capabilities they wish to check; while this flexibility is useful, it also makes it hard to guess what capability a certain command requires. We could make a solution that would work in a large majority of cases, but it wouldn't (and couldn't!) be absolutely correct in all circumstances, and since we're anal and we hate doing things halfway, we probably won't ever add this partial solution.

Why doesn't *Karma* seem to work for me?

Karma, by default, doesn't acknowledge karma updates. If you check the karma of whatever you increased/decreased, you'll note that your increment or decrement still took place. If you'd rather *Karma* acknowledge karma updates, change the *supybot.plugins.Karma.response* configuration variable to "On".

Why won't Supybot respond to private messages?

The most likely cause is that you are running your bot on the Freenode network. Around Sept. 2005, Freenode added a user mode which registered user could set that blocks private messages from unregistered users. So, the reason you aren't seeing a response from your Supybot is:

- Your Supybot is not registered with NickServ, you are registered, and you have set the +E user mode for yourself.
- or you have registered your Supybot with NickServ, you aren't registered, and your Supybot has the +E user mode set.

Can users with the "admin" capability change configuration?

Currently, no. Feel free to make your case to us as to why a certain configuration variable should only require the *admin* capability instead of the *owner* capability, and if we agree with you, we'll change it for the next release.

How can I make my Supybot log my IRC channel?

To log all the channels your Supybot is in, simply load the *ChannelLogger* plugin, which is included in the main distribution.

How do I find out channel modes?

I want to know who's an op in a certain channel, or who's voiced, or what the modes on the channel are. How do I do that?

Everything you need is kept in a *ChannelState* object in an *IrcState* object in the *Irc* object your plugin is given. To see the ops in a given channel, for instance, you would do this:

```
irc.state.channels['#channel'].ops
```

To see a dictionary mapping mode chars to values (if any), you would do this:

```
irc.state.channels['#channel'].modes
```

From there, things should be self-evident.

Can Supybot connect through a proxy server?

Supybot is not designed to be allowed to connect to an IRC server via a proxy server, however there are transparent proxy server helpers like tsocks that are designed to proxy-enable all network applications, and Supybot does work with these.

Why can't Supybot find the plugin I want to load?

Why does my bot say that 'No plugin "foo" exists.' when I try to load the foo plugin?

First, make sure you are typing the plugin name correctly. `@load foo` is not the same as `@load Foo` [1]. If that is not the problem,

I've found a bug, what do I do?

Submit your bug on GitHub through our project page.

Is Python installed?

I run Windows, and I'm not sure if Python is installed on my computer. How can I find out for sure?

Python isn't commonly installed by default on Windows computers. If you don't see it in your start menu somewhere, it's probably not installed.

The easiest way to find out if Python is installed is simply to download it and try to install it. If the installer complains, you probably already have it installed. If it doesn't, well, now you have Python installed.

[1] Yes, it used to be the same, but then we moved to using directories for plugins instead of a single file. Apparently, that makes a difference to Python.

CHAPTER 6

Getting Started with Supybot

6.1 Introduction

Ok, so you've decided to try out Supybot. That's great! The more people who use Supybot, the more people can submit bugs and help us to make it the best IRC bot in the world :)

You should have already read through our install document (if you had to manually install) before reading any further. Now we'll give you a whirlwind tour as to how you can get Supybot setup and use Supybot effectively.

6.2 Initial Setup

Now that you have Supybot installed, you'll want to get it running. The first thing you'll want to do is run supybot-wizard. Before running supybot-wizard, you should be in the directory in which you want your bot-related files to reside. The wizard will walk you through setting up a base config file for your Supybot. Once you've completed the wizard, you will have a config file called botname.conf. In order to get the bot running, run `supybot botname.conf`.

6.3 Listing Commands

Ok, so let's assume your bot connected to the server and joined the channels you told it to join. For now we'll assume you named your bot 'supybot' (you probably didn't, but it'll make it much clearer in the examples that follow to assume that you did). We'll also assume that you told it to join #channel (a nice generic name for a channel, isn't it? :)) So what do you do with this bot that you just made to join your channel? Try this in the channel:

```
supybot: list
```

Replacing 'supybot' with the actual name you picked for your bot, of course. Your bot should reply with a list of the plugins he currently has loaded. At least *Admin*, *Channel*, *Config*, *Misc*, *Owner*, and *User* should be there; if you used supybot-wizard to create your configuration file you may have many more plugins loaded. The list command can also be used to list the commands in a given plugin:

```
supybot: list Misc
```

will list all the commands in the *Misc* plugin. If you want to see the help for any command, just use the help command:

```
supybot: help help
supybot: help list
supybot: help load
```

Sometimes more than one plugin will have a given command; for instance, the "list" command exists in both the Misc and Config plugins (both loaded by default). List, in this case, defaults to the Misc plugin, but you may want to get the help for the list command in the Config plugin. In that case, you'll want to give your command like this:

```
supybot: help config list
```

Anytime your bot tells you that a given command is defined in several plugins, you'll want to use this syntax ("plugin command") to disambiguate which plugin's command you wish to call. For instance, if you wanted to call the Config plugin's list command, then you'd need to say:

```
supybot: config list
```

Rather than just 'list'.

6.4 Making Supybot Recognize You

If you ran the wizard, then it is almost certainly the case that you already added an owner user for yourself. If not, however, you can add one via the handy-dandy 'supybot-adduser' script. You'll want to run it while the bot is not running (otherwise it could overwrite supybot-adduser's changes to your user database before you get a chance to reload them). Just follow the prompts, and when it asks if you want to give the user any capabilities, say yes and then give yourself the 'owner' capability, restart the bot and you'll be ready to load some plugins!

Now, in order for the bot to recognize you as your owner user, you'll have to identify with the bot. Open up a query window in your irc client ('/query' should do it; if not, just know that you can't identify in a channel because it requires sending your password to the bot). Then type this:

```
help identify
```

And follow the instructions; the command you send will probably look like this, with 'myowneruser' and 'myuserpassword' replaced:

```
identify myowneruser myuserpassword
```

The bot will tell you that 'The operation succeeded' if you got the right name and password. Now that you're identified, you can do anything that requires any privilege: that includes all the commands in the Owner and Admin plugins, which you may want to take a look at (using the list and help commands, of course). One command in particular that you might want to use (it's from the User plugin) is the 'hostmask add' command: it lets you add a hostmask to your user record so the bot recognizes you by your hostmask instead of requiring you always to identify with it before it recognizes you. Use the 'help' command to see how this command works. Here's how I often use it:

```
hostmask add myuser [hostmask] mypassword
```

You may not have seen that '[hostmask]' syntax before. Supybot allows nested commands, which means that any command's output can be nested as an argument to another command. The hostmask command from the Misc plugin returns the hostmask of a given nick, but if given no arguments, it returns the hostmask of the person giving the command. So the command above adds the hostmask I'm currently using to my user's list of recognized hostmasks. I'm only required to give mypassword if I'm not already identified with the bot.

6.5 Loading Plugins

Let's take a look at loading other plugins. If you didn't use supybot-wizard, though, you might do well to try it before playing around with loading plugins yourself: each plugin has its own configure function that the wizard uses to setup the appropriate registry entries if the plugin requires any.

If you do want to play around with loading plugins, you're going to need to have the owner capability.

Remember earlier when I told you to try `help load`? That's the very command you'll be using. Basically, if you want to load, say, the Games plugin, then `load Games`. Simple, right? If you need a list of the plugins you can load, you'll have to list the directory the plugins are in (using whatever command is appropriate for your operating system, either 'ls' or 'dir').

6.6 Getting More From Your Supybot

Another command you might find yourself needing somewhat often is the 'more' command. The IRC protocol limits messages to 512 bytes, 60 or so of which must be devoted to some bookkeeping. Sometimes, however, Supybot wants to send a message that's longer than that. What it does, then, is break it into "chunks" and send the first one, following it with `(X more messages)` where X is how many more chunks there are. To get to these chunks, use the *more* command. One way to try is to look at the default value of *supybot.replies.genericNoCapability* – it's so long that it'll stretch across two messages:

```
<jemfinch|lambda> $config default
                  supybot.replies.genericNoCapability
<lambdaman> jemfinch|lambda: You're missing some capability
            you need. This could be because you actually
            possess the anti-capability for the capability
            that's required of you, or because the channel
            provides that anti-capability by default, or
            because the global capabilities include that
            anti-capability. Or, it could be because the
            channel or the global defaultAllow is set to
            False, meaning (1 more message)
<jemfinch|lambda> $more
<lambdaman> jemfinch|lambda: that no commands are allowed
            unless explicitly in your capabilities. Either
            way, you can't do what you want to do.
```

So basically, the bot keeps, for each person it sees, a list of "chunks" which are "released" one at a time by the *more* command. In fact, you can even get the more chunks for another user: if you want to see another chunk in the last command jemfinch gave, for instance, you would just say *more jemfinch* after which, his "chunks" now belong to you. So, you would just need to say *more* to continue seeing chunks from jemfinch's initial command.

6.7 Final Word

You should now have a solid foundation for using Supybot. You can use the *list* command to see what plugins your bot has loaded and what commands are in those plugins; you can use the 'help' command to see how to use a specific command, and you can use the 'more' command to continue a long response from the bot. With these three commands, you should have a strong basis with which to discover the rest of the features of Supybot!

Do be sure to read our other documentation and make use of the resources we provide for assistance; this website and, of course, #supybot on irc.freenode.net if you run into any trouble!

CHAPTER 7

Writing Your First Supybot Plugin

7.1 Introduction

Ok, so you want to write a plugin for Supybot. Good, then this is the place to be. We're going to start from the top (the highest level, where Supybot code does the most work for you) and move lower after that.

So have you used Supybot? If not, you need to go use it. This will help you understand crucial things like the way the various commands work and it is essential prior to embarking upon the plugin-development excursion detailed in the following pages. If you haven't used Supybot, come back to this document after you've used it for a while and gotten a feel for it.

So, now that we know you've used Supybot, we'll start getting into details. We'll go through this tutorial by actually writing a new plugin, named Random with just a few simple commands.

> Caveat: you'll need to have Supybot installed on the machine you intend to develop plugins on. This will not only allow you to test the plugins with a live bot, but it will also provide you with several nice scripts which aid the development of plugins. Most notably, it provides you with the supybot-plugin-create script which we will use in the next section... Creating a minimal plugin This section describes using the 'supybot-plugin-create' script to create a minimal plugin which we will enhance in later sections.

The recommended way to start writing a plugin is to use the wizard provided, **`supybot-plugin-create`**. Run this from within your local plugins directory, so we will be able to load the plugin and test it out.

It's very easy to follow, because basically all you have to do is answer three questions. Here's an example session:

```
[ddipaolo@quinn ../python/supybot]% supybot-plugin-create
What should the name of the plugin be? Random

Sometimes you'll want a callback to be threaded.  If its methods
(command or regexp-based, either one) will take a significant amount
of time to run, you'll want to thread them so they don't block the
entire bot.

Does your plugin need to be threaded? [y/n] n

What is your real name, so I can fill in the copyright and license
appropriately? Daniel DiPaolo

Your new plugin template is in the Random directory.
```

It's that simple! Well, that part of making the minimal plugin is that simple. You should now have a directory with a few files in it, so let's take a look at each of those files and see what they're used for.

7.2 README.txt

In `README.txt` you put exactly what the boilerplate text says to put in there:

> Insert a description of your plugin here, with any notes, etc. about using it.

A brief overview of exactly what the purpose of the plugin is supposed to do is really all that is needed here. Also, if this plugin requires any third-party Python modules, you should definitely mention those here. You don't have to describe individual commands or anything like that, as those are defined within the plugin code itself as you'll see later. You also don't need to acknowledge any of the developers of the plugin as those too are handled elsewhere.

For our Random plugin, let's make `README.txt` say this:

> This plugin contains commands relating to random numbers, and includes: a simple random number generator, the ability to pick a random number from within a range, a command for returning a random sampling from a list of items, and a simple dice roller.

And now you know what's in store for the rest of this tutorial, we'll be writing all of that in one Supybot plugin, and you'll be surprised at just how simple it is!

7.3 __init__.py

The next file we'll look at is `__init__.py`. If you're familiar with the Python import mechanism, you'll know what this file is for. If you're not, think of it as sort of the "glue" file that pulls all the files in this directory together when you load the plugin. It's also where there are a few administrative items live that you really need to maintain.

Let's go through the file. For the first 30 lines or so, you'll see the copyright notice that we use for our plugins, only with your name in place (as prompted in **`supybot-plugin-create`**). Feel free to use whatever license you choose, we don't feel particularly attached to the boilerplate code so it's yours to license as you see fit even if you don't modify it. For our example, we'll leave it as is.

The plugin docstring immediately follows the copyright notice and it (like `README.txt`) tells you precisely what it should contain:

> Add a description of the plugin (to be presented to the user inside the wizard) here. This should describe *what* the plugin does.

The "wizard" that it speaks of is the **`supybot-wizard`** script that is used to create working Supybot config file. I imagine that in meeting the prerequisite of "using a Supybot" first, most readers will have already encountered this script. Basically, if the user selects to look at this plugin from the list of plugins to load, it prints out that description to let the user know what it does, so make sure to be clear on what the purpose of the plugin is. This should be an abbreviated version of what we put in our `README.txt`, so let's put this:

```
Provides a number of commands for selecting random things.
```

Next in `__init__.py` you see a few imports which are necessary, and then four attributes that you need to modify for your bot and preferably keep up with as you develop it: `__version__`, `__author__`, `__contributors__`, `__url__`.

`__version__` is just a version string representing the current working version of the plugin, and can be anything you want. If you use some sort of RCS, this would be a good place to have it automatically increment the version string for any time you edit any of the files in this directory. We'll just make ours "0.1".

`__author__` should be an instance of the `supybot.Author` class. A `supybot.Author` is simply created by giving it a full name, a short name (preferably IRC nick), and an e-mail address (all of these are optional, though at least the second one is expected). So, for example, to create my Author user (though I get to cheat and use supybot.authors.strike since I'm a main dev, muahaha), I would do:

```
__author__ = supybot.Author('Daniel DiPaolo', 'Strike',
                            'somewhere@someplace.xxx')
```

Keep this in mind as we get to the next item...

__contributors__ is a dictionary mapping supybot.Author instances to lists of things they contributed. If someone adds a command named foo to your plugin, the list for that author should be ["foo"], or perhaps even ["added foo command"]. The main author shouldn't be referenced here, as it is assumed that everything that wasn't contributed by someone else was done by the main author. For now we have no contributors, so we'll leave it blank.

Lastly, the __url__ attribute should just reference the download URL for the plugin. Since this is just an example, we'll leave this blank.

The rest of __init__.py really shouldn't be touched unless you are using third-party modules in your plugin. If you are, then you need to take special note of the section that looks like this:

```
import config
import plugin
reload(plugin) # In case we're being reloaded.
# Add more reloads here if you add third-party modules and want them
# to be reloaded when this plugin is reloaded.  Don't forget to
# import them as well!
```

As the comment says, this is one place where you need to make sure you import the third-party modules, and that you call reload() on them as well. That way, if we are reloading a plugin on a running bot it will actually reload the latest code. We aren't using any third-party modules, so we can just leave this bit alone.

We're almost through the "boring" part and into the guts of writing Supybot plugins, let's take a look at the next file.

7.4 config.py

config.py is, unsurprisingly, where all the configuration stuff related to your plugin goes. If you're not familiar with Supybot's configuration system, I recommend reading the config tutorial before going any further with this section.

So, let's plow through config.py line-by-line like we did the other files.

Once again, at the top is the standard copyright notice. Again, change it to how you see fit.

Then, some standard imports which are necessary.

Now, the first peculiar thing we get to is the configure function. This function is what is called by the supybot-wizard whenever a plugin is selected to be loaded. Since you've used the bot by now (as stated on the first page of this tutorial as a prerequisite), you've seen what this script does to configure plugins. The wizard allows the bot owner to choose something different from the default plugin config values without having to do it through the bot (which is still not difficult, but not as easy as this). Also, note that the advanced argument allows you to differentiate whether or not the person configuring this plugin considers himself an advanced Supybot user. Our plugin has no advanced features, so we won't be using it.

So, what exactly do we do in this configure function for our plugin? Well, for the most part we ask questions and we set configuration values. You'll notice the import line with supybot.questions in it. That provides some nice convenience functions which are used to (you guessed it) ask questions. The other line in there is the conf.registerPlugin line which registers our plugin with the config and allows us to create configuration values for the plugin. You should leave these two lines in even if you don't have anything else to put in here. For the vast majority of plugins, you can leave this part as is, so we won't go over how to write plugin configuration functions here (that will be handled in a separate article). Our plugin won't be using much configuration, so we'll leave this as is.

Next, you'll see a line that looks very similar to the one in the configure function. This line is used not only to register the plugin prior to being called in configure, but also to store a bit of an alias to the plugin's config group to make

things shorter later on. So, this line should read:

```
Random = conf.registerPlugin('Random')
```

Now we get to the part where we define all the configuration groups and variables that our plugin is to have. Again, many plugins won't require any configuration so we won't go over it here, but in a separate article dedicated to sprucing up your config.py for more advanced plugins. Our plugin doesn't require any config variables, so we actually don't need to make any changes to this file at all.

Configuration of plugins is handled in depth at the Advanced Plugin Config Tutorial

7.5 plugin.py

Here's the moment you've been waiting for, the overview of plugin.py and how to make our plugin actually do stuff.

At the top, same as always, is the standard copyright block to be used and abused at your leisure.

Next, some standard imports. Not all of them are used at the moment, but you probably will use many (if not most) of them, so just let them be. Since we'll be making use of Python's standard 'random' module, you'll need to add the following line to the list of imports:

```
import random
```

Now, the plugin class itself. What you're given is a skeleton: a simple subclass of callbacks.Plugin for you to start with. The only real content it has is the boilerplate docstring, which you should modify to reflect what the boilerplate text says - it should be useful so that when someone uses the plugin help command to determine how to use this plugin, they'll know what they need to do. Ours will read something like:

```
"""This plugin provides a few random number commands and some
commands for getting random samples.  Use the "seed" command to seed
the plugin's random number generator if you like, though it is
unnecessary as it gets seeded upon loading of the plugin.  The
"random" command is most likely what you're looking for, though
there are a number of other useful commands in this plugin.  Use
'list random' to check them out.  """
```

It's basically a "guide to getting started" for the plugin. Now, to make the plugin do something. First of all, to get any random numbers we're going to need a random number generator (RNG). Pretty much everything in our plugin is going to use it, so we'll define it in the constructor of our plugin, __init__. Here we'll also seed it with the current time (standard practice for RNGs). Here's what our __init__ looks like:

```
def __init__(self, irc):
    self.__parent = super(Random, self)
    self.__parent.__init__(irc)
    self.rng = random.Random()    # create our rng
    self.rng.seed()    # automatically seeds with current time
```

Now, the first two lines may look a little daunting, but it's just administrative stuff required if you want to use a custom __init__. If we didn't want to do so, we wouldn't have to, but it's not uncommon so I decided to use an example plugin that did. For the most part you can just copy/paste those lines into any plugin you override the __init__ for and just change them to use the plugin name that you are working on instead.

So, now we have a RNG in our plugin, let's write a command to get a random number. We'll start with a simple command named random that just returns a random number from our RNG and takes no arguments. Here's what that looks like:

```
def random(self, irc, msg, args):
    """takes no arguments

    Returns the next random number from the random number generator.
    """
    irc.reply(str(self.rng.random()))
random = wrap(random)
```

And that's it. Now here are the important points.

First and foremost, all plugin commands must have all-lowercase function names. If they aren't all lowercase they won't show up in a plugin's list of commands (nor will they be useable in general). If you look through a plugin and see a function that's not in all lowercase, it is not a plugin command. Chances are it is a helper function of some sort, and in fact using capital letters is a good way of assuring that you don't accidentally expose helper functions to users as commands.

You'll note the arguments to this class method are (self, irc, msg, args). This is what the argument list for all methods that are to be used as commands must start with. If you wanted additional arguments, you'd append them onto the end, but since we take no arguments we just stop there. I'll explain this in more detail with our next command, but it is very important that all plugin commands are class methods that start with those four arguments exactly as named.

Next, in the docstring there are two major components. First, the very first line dictates the argument list to be displayed when someone calls the help command for this command (i.e., help random). Then you leave a blank line and start the actual help string for the function. Don't worry about the fact that it's tabbed in or anything like that, as the help command normalizes it to make it look nice. This part should be fairly brief but sufficient to explain the function and what (if any) arguments it requires. Remember that this should fit in one IRC message which is typically around a 450 character limit.

Then we have the actual code body of the plugin, which consists of a single line: irc.reply(str(self.rng.random())). The irc.reply function issues a reply to wherever the PRIVMSG it received the command from with whatever text is provided. If you're not sure what I mean when I say "wherever the PRIVMSG it received the command from", basically it means: if the command is issued in a channel the response is sent in the channel, and if the command is issued in a private dialog the response is sent in a private dialog. The text we want to display is simply the next number from our RNG (self.rng). We get that number by calling the random function, and then we str it just to make sure it is a nice printable string.

Lastly, all plugin commands must be 'wrap'ed. What the wrap function does is handle argument parsing for plugin commands in a very nice and very powerful way. With no arguments, we simply need to just wrap it. For more in-depth information on using wrap check out the wrap tutorial (The astute Python programmer may note that this is very much like a decorator, and that's precisely what it is. However, we developed this before decorators existed and haven't changed the syntax due to our earlier requirement to stay compatible with Python 2.3. As we now require Python 2.6 or greater, this may eventually change to support work via decorators.)

Now let's create a command with some arguments and see how we use those in our plugin commands. Let's allow the user to seed our RNG with their own seed value. We'll call the command seed and take just the seed value as the argument (which we'll require be a floating point value of some sort, though technically it can be any hashable object). Here's what this command looks like:

```
def seed(self, irc, msg, args, seed):
    """<seed>

    Sets the internal RNG's seed value to <seed>.  <seed> must be a
    floating point number.
    """
    self.rng.seed(seed)
    irc.replySuccess()
seed = wrap(seed, ['float'])
```

You'll notice first that argument list now includes an extra argument, seed. If you read the wrap tutorial mentioned above, you should understand how this arg list gets populated with values. Thanks to wrap we don't have to worry about type-checking or value-checking or anything like that. We just specify that it must be a float in the wrap portion and we can use it in the body of the function.

Of course, we modify the docstring to document this function. Note the syntax on the first line. Arguments go in <> and optional arguments should be surrounded by [] (we'll demonstrate this later as well).

The body of the function should be fairly straightforward to figure out, but it introduces a new function - irc.replySuccess. This is just a generic "I succeeded" command which responds with whatever the bot owner has configured to be the success response (configured in supybot.replies.success). Note that we don't do any error-checking in the plugin, and that's because we simply don't have to. We are guaranteed that seed will be a float and so the call to our RNG's seed is guaranteed to work.

Lastly, of course, the wrap call. Again, read the wrap tutorial for fuller coverage of its use, but the basic premise is that the second argument to wrap is a list of converters that handles argument validation and conversion and it then assigns values to each argument in the arg list after the first four (required) arguments. So, our seed argument gets a float, guaranteed.

With this alone you'd be able to make some pretty usable plugin commands, but we'll go through two more commands to introduce a few more useful ideas. The next command we'll make is a sample command which gets a random sample of items from a list provided by the user:

```
def sample(self, irc, msg, args, n, items):
    """<number of items> <item1> [<item2> ...]

    Returns a sample of the <number of items> taken from the remaining
    arguments.  Obviously <number of items> must be less than the number
    of arguments given.
    """
    if n > len(items):
        irc.error('<number of items> must be less than the number '
                  'of arguments.')
        return
    sample = self.rng.sample(items, n)
    sample.sort()
    irc.reply(utils.str.commaAndify(sample))
sample = wrap(sample, ['int', many('anything')])
```

This plugin command introduces a few new things, but the general structure should look fairly familiar by now. You may wonder why we only have two extra arguments when obviously this plugin can accept any number of arguments. Well, using wrap we collect all of the remaining arguments after the first one into the items argument. If you haven't caught on yet, wrap is really cool and extremely useful.

Next of course is the updated docstring. Note the use of [] to denote the optional items after the first item.

The body of the plugin should be relatively easy to read. First we check and make sure that n (the number of items the user wants to sample) is not larger than the actual number of items they gave. If it does, we call irc.error with the error message you see. irc.error is kind of like irc.replySuccess only it gives an error message using the configured error format (in supybot.replies.error). Otherwise, we use the sample function from our RNG to get a sample, then we sort it, and we reply with the 'utils.str.commaAndify'ed version. The utils.str.commaAndify function basically takes a list of strings and turns it into "item1, item2, item3, item4, and item5" for an arbitrary length. More details on using the utils module can be found in the utils tutorial.

Now for the last command that we will add to our plugin.py. This last command will allow the bot users to roll an arbitrary n-sided die, with as many sides as they so choose. Here's the code for this command:

```
def diceroll(self, irc, msg, args, n):
    """[<number of sides>]
```

```
    Rolls a die with <number of sides> sides.  The default number of sides
    is 6.
    """
    s = 'rolls a %s' % self.rng.randrange(1, n)
    irc.reply(s, action=True)
diceroll = wrap(diceroll, [additional(('int', 'number of sides'), 6)])
```

The only new thing learned here really is that the irc.reply method accepts an optional argument action, which if set to True makes the reply an action instead. So instead of just crudely responding with the number, instead you should see something like * supybot rolls a 5. You'll also note that it uses a more advanced wrap line than we have used to this point, but to learn more about wrap, you should refer to the wrap tutorial

And now that we're done adding plugin commands you should see the boilerplate stuff at the bottom, which just consists of:

```
Class = Random
```

And also some vim modeline stuff. Leave these as is, and we're finally done with plugin.py!

7.6 test.py

Now that we've gotten our plugin written, we want to make sure it works. Sure, an easy way to do a somewhat quick check is to start up a bot, load the plugin, and run a few commands on it. If all goes well there, everything's probably okay. But, we can do better than "probably okay". This is where written plugin tests come in. We can write tests that not only assure that the plugin loads and runs the commands fine, but also that it produces the expected output for given inputs. And not only that, we can use the nifty supybot-test script to test the plugin without even having to have a network connection to connect to IRC with and most certainly without running a local IRC server.

The boilerplate code for test.py is a good start. It imports everything you need and sets up RandomTestCase which will contain all of our tests. Now we just need to write some test methods. I'll be moving fairly quickly here just going over very basic concepts and glossing over details, but the full plugin test authoring tutorial has much more detail to it and is recommended reading after finishing this tutorial.

Since we have four commands we should have at least four test methods in our test case class. Typically you name the test methods that simply checks that a given command works by just appending the command name to test. So, we'll have testRandom, testSeed, testSample, and testDiceRoll. Any other methods you want to add are more free-form and should describe what you're testing (don't be afraid to use long names).

First we'll write the testRandom method:

```
def testRandom(self):
    # difficult to test, let's just make sure it works
    self.assertNotError('random')
```

Since we can't predict what the output of our random number generator is going to be, it's hard to specify a response we want. So instead, we just make sure we don't get an error by calling the random command, and that's about all we can do.

Next, testSeed. In this method we're just going to check that the command itself functions. In another test method later on we will check and make sure that the seed produces reproducible random numbers like we would hope it would, but for now we just test it like we did random in 'testRandom':

```
def testSeed(self):
    # just make sure it works
    self.assertNotError('seed 20')
```

Now for testSample. Since this one takes more arguments it makes sense that we test more scenarios in this one. Also this time we have to make sure that we hit the error that we coded in there given the right conditions:

```
def testSample(self):
    self.assertError('sample 20 foo')
    self.assertResponse('sample 1 foo', 'foo')
    self.assertRegexp('sample 2 foo bar', '... and ...')
    self.assertRegexp('sample 3 foo bar baz', '..., ..., and ...')
```

So first we check and make sure trying to take a 20-element sample of a 1-element list gives us an error. Next we just check and make sure we get the right number of elements and that they are formatted correctly when we give 1, 2, or 3 element lists.

And for the last of our basic "check to see that it works" functions, testDiceRoll:

```
def testDiceRoll(self):
    self.assertActionRegexp('diceroll', 'rolls a \d')
```

We know that diceroll should return an action, and that with no arguments it should roll a single-digit number. And that's about all we can test reliably here, so that's all we do.

Lastly, we wanted to check and make sure that seeding the RNG with seed actually took effect like it's supposed to. So, we write another test method:

```
def testSeedActuallySeeds(self):
    # now to make sure things work repeatably
    self.assertNotError('seed 20')
    m1 = self.getMsg('random')
    self.assertNotError('seed 20')
    m2 = self.getMsg('random')
    self.failUnlessEqual(m1, m2)
    m3 = self.getMsg('random')
    self.failIfEqual(m2, m3)
```

So we seed the RNG with 20, store the message, and then seed it at 20 again. We grab that message, and unless they are the same number when we compare the two, we fail. And then just to make sure our RNG is producing random numbers, we get another random number and make sure it is distinct from the prior one.

7.7 Conclusion

You are now very well-prepared to write Supybot plugins. Now for a few words of wisdom with regards to Supybot plugin-writing.

- Read other people's plugins, especially the included plugins and ones by the core developers. We (the Supybot dev team) can't possibly document all the awesome things that Supybot plugins can do, but we try. Nevertheless there are some really cool things that can be done that aren't very well-documented.
- Hack new functionality into existing plugins first if writing a new plugin is too daunting.
- Come ask us questions in #supybot on Freenode or OFTC. Going back to the first point above, the developers themselves can help you even more than the docs can (though we prefer you read the docs first).
- Share your plugins with the world and make Supybot all that more attractive for other users so they will want to write their plugins for Supybot as well.
- Read, read, read all the documentation.
- And of course, have fun writing your plugins.

CHAPTER 8

Style Guidelines

Note: Code not following these style guidelines fastidiously is likely (*very* likely) not to be accepted into the Supybot core.

- Read **PEP 8** (Guido's Style Guide) and know that we use almost all the same style guidelines.
- Maximum line length is 79 characters. 78 is a safer bet, though. This is **NON-NEGOTIABLE**. Your code will not be accepted while you are violating this guidline.
- Identation is 4 spaces per level. No tabs. This also is **NON-NEGOTIABLE**. Your code, again, will *never* be accepted while you have literal tabs in it.
- Single quotes are used for all string literals that aren't docstrings. They're just easier to type.
- Triple double quotes (`"""`) are always used for docstrings.
- Raw strings (`r''` or `r""`) should be used for regular expressions.
- Spaces go around all operators (except around = in default arguments to functions) and after all commas (unless doing so keeps a line within the 79 character limit).
- Functions calls should look like `foo(bar(baz(x), y))`. They should not look like `foo (bar (baz (x), y))`, or like `foo(bar(baz(x), y) )` or like anything else. I hate extraneous spaces.
- Class names are StudlyCaps. Method and function names are camelCaps (StudlyCaps with an initial lowercase letter). If variable and attribute names can maintain readability without being camelCaps, then they should be entirely in lowercase, otherwise they should also use camelCaps. Plugin names are StudlyCaps.
- Imports should always happen at the top of the module, one import per line (so if imports need to be added or removed later, it can be done easily).
- Unless absolutely required by some external force, imports should be ordered by the string length of the module imported. I just think it looks prettier.
- A blank line should be between all consecutive method declarations in a class definition. Two blank lines should be between all consecutive class definitions in a file. Comments are even better than blank lines for separating classes.
- Database filenames should generally begin with the name of the plugin and the extension should be 'db'. plugins.DBHandler does this already.
- Whenever creating a file descriptor or socket, keep a reference around and be sure to close it. There should be no code like this:

```
s = urllib2.urlopen('url').read()
```

  Instead, do this:

```
fd = urllib2.urlopen('url')
try:
    s = fd.read()
finally:
    fd.close()
```

This is to be sure the bot doesn't leak file descriptors.

- All plugin files should include a docstring decsribing what the plugin does. This docstring will be returned when the user is configuring the plugin. All plugin classes should also include a docstring describing how to do things with the plugin; this docstring will be returned when the user requests help on a plugin name.
- Method docstrings in classes deriving from callbacks.Privmsg should include an argument list as their first line, and after that a blank line followed by a longer description of what the command does. The argument list is used by the `syntax` command, and the longer description is used by the `help` command.
- Whenever joining more than two strings, use string interpolation, not addition:

```
s = x + y + z # Bad.
s = '%s%s%s' % (x, y, z) # Good.
s = ''.join([x, y, z]) # Best, but not as general.
```

This has to do with efficiency; the intermediate string x+y is made (and thus copied) before x+y+z is made, so it's less efficient. People who use string concatenation in a for loop will be swiftly kicked in the head.

- When writing strings that have formatting characters in them, don't use anything but `%s` unless you absolutely must. In particular, `%d` should never be used, it's less general than `%s` and serves no useful purpose. If you got the `%d` wrong, you'll get an exception that says, "foo instance can't be converted to an integer." But if you use `%s`, you'll get to see your nice little foo instance, if it doesn't convert to a string cleanly, and if it does convert cleanly, you'll get to see what you expect to see. Basically, `%d` just sucks.
- As a corrolary to the above, note that sometimes `%f` is used, but on when floats need to be formatted, e.g., `%.2f`.
- Use the log module to its fullest; when you need to print some values to debug, use self.log.debug to do so, and leave those statements in the code (commented out) so they can later be re-enabled. Remember that once code is buggy, it tends to have more bugs, and you'll probably need those print statements again.
- While on the topic of logs, note that we do not use % (i.e., str.__mod__) with logged strings; we simple pass the format parameters as additional arguments. The reason is simple: the logging module supports it, and it's cleaner (fewer tokens/glyphs) to read.
- While still on the topic of logs, it's also important to pick the appropriate log level for given information.
  - DEBUG: Appropriate to tell a programmer *how* we're doing something (i.e., debugging printfs, basically). If you're trying to figure out why your code doesn't work, DEBUG is the new printf – use that, and leave the statements in your code.
  - INFO: Appropriate to tell a user *what* we're doing, when what we're doing isn't important for the user to pay attention to. A user who likes to keep up with things should enjoy watching our logging at the INFO level; it shouldn't be too low-level, but it should give enough information that it keeps him relatively interested at peak times.
  - WARNING: Appropriate to tell a user when we're doing something that he really ought to pay attention to. Users should see WARNING and think, "Hmm, should I tell the Supybot developers about this?" Later, he should decide not to, but it should give the user a moment to pause and think about what's actually happening with his bot.
  - ERROR: Appropriate to tell a user when something has gone wrong. Uncaught exceptions are ERRORs. Conditions that we absolutely want to hear about should be errors. Things that should *scare* the user should be errors.

  - CRITICAL: Not really appropriate. I can think of no absolutely critical issue yet encountered in Supybot; the only possible thing I can imagine is to notify the user that the partition on which Supybot is running has filled up. That would be a CRITICAL condition, but it would also be hard to log :)

- All plugins should have test cases written for them. Even if it doesn't actually test anything but just exists, it's good to have the test there so there's a place to add more tests later (and so we can be sure that all plugins are adequately documented; PluginTestCase checks that every command has documentation)
- All uses of eval() that expect to get integrated in Supybot must be approved by jemfinch, no exceptions. Chances are, it won't be accepted. Have you looked at utils.safeEval?
- SQL table names should be all-lowercase and include underscores to separate words. This is because SQL itself is case-insensitive. This doesn't change, however the fact that variable/member names should be camel case.
- SQL statements in code should put SQL words in ALL CAPS:

```
"""SELECT quote FROM quotes ORDER BY random() LIMIT 1"""
```

  This makes SQL significantly easier to read.

- Common variable names
  - L => an arbitrary list.
  - t => an arbitrary tuple.
  - x => an arbitrary float.
  - s => an arbitrary string.
  - f => an arbitrary function.
  - p => an arbitrary predicate.
  - i,n => an arbitrary integer.
  - cb => an arbitrary callback.
  - db => a database handle.
  - fd => a file-like object.
  - msg => an ircmsgs.IrcMsg object.
  - irc => an irclib.Irc object (or proxy)
  - nick => a string that is an IRC nick.
  - channel => a string that is an IRC channel.
  - hostmask => a string that is a user's IRC prefix.

  When the semantic functionality (that is, the "meaning" of a variable is obvious from context), one of these names should be used. This just makes it easier for people reading our code to know what a variable represents without scouring the surrounding code.

- Multiple variable assignments should always be surrounded with parentheses – i.e., if you're using the partition function, then your assignment statement should look like:

```
(good, bad) = partition(p, L)
```

  The parentheses make it obvious that you're doing a multiple assignment, and that's important because I hate reading code and wondering where a variable came from.

CHAPTER 9

Using Supybot's utils module

Supybot provides a wealth of utilities for plugin writers in the supybot.utils module, this tutorial describes these utilities and shows you how to use them.

9.1 str.py

9.1.1 The Format Function

The supybot.utils.str module provides a bunch of utility functions for handling string values. This section contains a quick rundown of all of the functions available, along with descriptions of the arguments they take. First and foremost is the format function, which provides a lot of capability in just one function that uses string-formatting style to accomplish a lot. So much so that it gets its own section in this tutorial. All other functions will be in other sections. format takes several arguments - first, the format string (using the format characters described below), and then after that, each individual item to be formatted. Do not attempt to use the % operator to do the formatting because that will fall back on the normal string formatting operator. The format function uses the following string formatting characters.

- % - literal `%`
- i - integer
- s - string
- f - float
- r - repr
- b - form of the verb `to be` (takes an int)
- h - form of the verb `to have` (takes an int)
- L - commaAndify (takes a list of strings or a tuple of ([strings], and))
- p - pluralize (takes a string)
- q - quoted (takes a string)
- n - n items (takes a 2-tuple of (n, item) or a 3-tuple of (n, between, item))
- t - time, formatted (takes an int)
- u - url, wrapped in braces

Here are a few examples to help elaborate on the above descriptions:

```
>>> format("Error %q has been reported %n.  For more information, see %u.",
           "AttributeError", (5, "time"), "http://supybot.com")

'Error "AttributeError" has been reported 5 times.  For more information,
 see <http://supybot.com>.'

>>> i = 4
>>> format("There %b %n at this time.  You are only allowed %n at any given
            time", i, (i, "active", "thread"), (5, "active", "thread"))
'There are 4 active threads at this time.  You are only allowed 5 active
 threads at any given time'

>>> i = 1
>>> format("There %b %n at this time.  You are only allowed %n at any given
            time", i, (i, "active", "thread"), (5, "active", "thread"))
 'There is 1 active thread at this time.  You are only allowed 5 active
  threads at any given time'

>>> ops = ["foo", "bar", "baz"]
>>> format("The following %n %h the %s capability: %L", (len(ops), "user"),
            len(ops), "op", ops)
'The following 3 users have the op capability: foo, bar, and baz'
```

As you can see, you can combine all sorts of combinations of formatting strings into one. In fact, that was the major motivation behind format. We have specific functions that you can use individually for each of those formatting types, but it became much easier just to use special formatting chars and the format function than concatenating a bunch of strings that were the result of other utils.str functions.

9.1.2 The Other Functions

These are the functions that can't be handled by format. They are sorted in what I perceive to be the general order of usefulness (and I'm leaving the ones covered by format for the next section).

- ellipsisify(s, n) - Returns a shortened version of a string. Produces up to the first n chars at the nearest word boundary.
  - s: the string to be shortened
  - n: the number of characters to shorten it to
- perlReToPythonRe(s) - Converts a Perl-style regexp (e.g., "/abcd/i" or "m/abcd/i") to an actual Python regexp (an re object)
  - s: the regexp string
- perlReToReplacer(s) - converts a perl-style replacement regexp (eg, "s/foo/bar/g") to a Python function that performs such a replacement
  - s: the regexp string
- dqrepr(s) - Returns a repr() of s guaranteed to be in double quotes. (Double Quote Repr)
  - s: the string to be double-quote repr()'ed
- toBool(s) - Determines whether or not a string means True or False and returns the appropriate boolean value. True is any of "true", "on", "enable", "enabled", or "1". False is any of "false", "off", "disable", "disabled", or "0".
  - s: the string to determine the boolean value for

- rsplit(s, sep=None, maxsplit=-1) - functionally the same as str.split in the Python standard library except splitting from the right instead of the left. Python 2.4 has str.rsplit (which this function defers to for those versions >= 2.4), but Python 2.3 did not.
  - s: the string to be split
  - sep: the separator to split on, defaults to whitespace
  - maxsplit: the maximum number of splits to perform, -1 splits all possible splits.
- normalizeWhitespace(s) - reduces all multi-spaces in a string to a single space
  - s: the string to normalize
- depluralize(s) - the opposite of pluralize
  - s: the string to depluralize
- unCommaThe(s) - Takes a string of the form "foo, the" and turns it into "the foo"
  - s: string, the
- distance(s, t) - computes the levenshtein distance (or "edit distance") between two strings
  - s: the first string
  - t: the second string
- soundex(s, length=4) - computes the soundex for a given string
  - s: the string to compute the soundex for
  - length: the length of the soundex to generate
- matchCase(s1, s2) - Matches the case of the first string in the second string.
  - s1: the first string
  - s2: the string which will be made to match the case of the first

9.1.3 The Commands Format Already Covers

These commands aren't necessary because you can achieve them more easily by using the format command, but they exist if you decide you want to use them anyway though it is greatly discouraged for general use.

- commaAndify(seq, comma=",", And="and") - transforms a list of items into a comma separated list with an "and" preceding the last element. For example, ["foo", "bar", "baz"] becomes "foo, bar, and baz". Is smart enough to convert two-element lists to just "item1 and item2" as well.
  - seq: the sequence of items (don't have to be strings, but need to be 'str()'-able)
  - comma: the character to use to separate the list
  - And: the word to use before the last element
- pluralize(s) - Returns the plural of a string. Put any exceptions to the general English rules of pluralization in the plurals dictionary in supybot.utils.str.
  - s: the string to pluralize
- nItems(n, item, between=None) - returns a string that describes a given number of an item (with any string between the actual number and the item itself), handles pluralization with the pluralize function above. Note that the arguments here are in a different order since between is optional.
  - n: the number of items
  - item: the type of item

  - between: the optional string that goes between the number and the type of item
- quoted(s) - Returns the string surrounded by double-quotes.
  - s: the string to quote
- be(i) - Returns the proper form of the verb "to be" based on the number provided (be(1) is "is", be(anything else) is "are")
  - i: the number of things that "be"
- has(i) - Returns the proper form of the verb "to have" based on the number provided (has(1) is "has", has(anything else) is "have")
  - i: the number of things that "has"

9.2 structures.py

9.2.1 Intro

This module provides a number of useful data structures that aren't found in the standard Python library. For the most part they were created as needed for the bot and plugins themselves, but they were created in such a way as to be of general use for anyone who needs a data structure that performs a like duty. As usual in this document, I'll try and order these in order of usefulness, starting with the most useful.

9.2.2 The queue classes

The structures module provides two general-purpose queue classes for you to use. The "queue" class is a robust full-featured queue that scales up to larger sized queues. The "smallqueue" class is for queues that will contain fewer (less than 1000 or so) items. Both offer the same common interface, which consists of:

- a constructor which will optionally accept a sequence to start the queue off with
- enqueue(item) - adds an item to the back of the queue
- dequeue() - removes (and returns) the item from the front of the queue
- peek() - returns the item from the front of the queue without removing it
- reset() - empties the queue entirely

In addition to these general-use queue classes, there are two other more specialized queue classes as well. The first is the "TimeoutQueue" which holds a queue of items until they reach a certain age and then they are removed from the queue. It features the following:

- TimeoutQueue(timeout, queue=None) - you must specify the timeout (in seconds) in the constructor. Note that you can also optionally pass it a queue which uses any implementation you wish to use whether it be one of the above (queue or smallqueue) or if it's some custom queue you create that implements the same interface. If you don't pass it a queue instance to use, it will build its own using smallqueue.
  - reset(), enqueue(item), dequeue() - all same as above queue classes
  - setTimeout(secs) - allows you to change the timeout value

And for the final queue class, there's the "MaxLengthQueue" class. As you may have guessed, it's a queue that is capped at a certain specified length. It features the following:

- MaxLengthQueue(length, seq=()) - the constructor naturally requires that you set the max length and it allows you to optionally pass in a sequence to be used as the starting queue. The underlying implementation is actually the queue from before.

  - enqueue(item) - adds an item onto the back of the queue and if it would push it over the max length, it dequeues the item on the front (it does not return this item to you)
  - all the standard methods from the queue class are inherited for this class

9.2.3 The Other Structures

The most useful of the other structures is actually very similar to the "MaxLengthQueue". It's the "RingBuffer", which is essentially a MaxLengthQueue which fills up to its maximum size and then circularly replaces the old contents as new entries are added instead of dequeuing. It features the following:

- RingBuffer(size, seq=()) - as with the MaxLengthQueue you specify the size of the RingBuffer and optionally give it a sequence.
  - append(item) - adds item to the end of the buffer, pushing out an item from the front if necessary
  - reset() - empties out the buffer entirely
  - resize(i) - shrinks/expands the RingBuffer to the size provided
  - extend(seq) - append the items from the provided sequence onto the end of the RingBuffer

The next data structure is the TwoWayDictionary, which as the name implies is a dictionary in which key-value pairs have mappings going both directions. It features the following:

- TwoWayDictionary(seq=(), **kwargs) - Takes an optional sequence of (key, value) pairs as well as any key=value pairs specified in the constructor as initial values for the two-way dict.
  - other than that, no extra features that a normal Python dict doesn't already offer with the exception that any (key, val) pair added to the dict is also added as (val, key) as well, so the mapping goes both ways. Elements are still accessed the same way you always do with Python 'dict's.

There is also a MultiSet class available, but it's very unlikely that it will serve your purpose, so I won't go into it here. The curious coder can go check the source and see what it's all about if they wish (it's only used once in our code, in the Relay plugin).

9.3 web.py

The web portion of Supybot's utils module is mainly used for retrieving data from websites but it also has some utility functions pertaining to HTML and email text as well. The functions in web are listed below, once again in order of usefulness.

- getUrl(url, size=None, headers=None) - gets the data at the URL provided and returns it as one large string
  - url: the location of the data to be retrieved or a urllib2.Request object to be used in the retrieval
  - size: the maximum number of bytes to retrieve, defaults to None, meaning that it is to try to retrieve all data
  - headers: a dictionary mapping header types to header data
- getUrlFd(url, headers=None) - returns a file-like object for a url
  - url: the location of the data to be retrieved or a urllib2.Request object to be used in the retrieval
  - headers: a dictionary mapping header types to header data
- htmlToText(s, tagReplace=" ") - strips out all tags in a string of HTML, replacing them with the specified character
  - s: the HTML text to strip the tags out of

  - tagReplace: the string to replace tags with
- strError(e) - pretty-printer for web exceptions, returns a descriptive string given a web-related exception
  - e: the exception to pretty-print
- mungeEmail(s) - a naive e-mail obfuscation function, replaces "@" with "AT" and "." with "DOT"
  - s: the e-mail address to obfuscate
- getDomain(url) - returns the domain of a URL - url: the URL in question

9.4 The Best of the Rest

9.4.1 Intro

Rather than document each of the remaining portions of the supybot.utils module, I've elected to just pick out the choice bits from specific parts and document those instead. Here they are, broken out by module name.

9.4.2 supybot.utils.file - file utilities

- touch(filename) - updates the access time of a file by opening it for writing and immediately closing it
- mktemp(suffix="") - creates a decent random string, suitable for a temporary filename with the given suffix, if provided
- the AtomicFile class - used for files that need to be atomically written, i.e., if there's a failure the original file remains unmodified. For more info consult file.py in src/utils

9.4.3 supybot.utils.gen - general utilities

- timeElapsed(elapsed, [lots of optional args]) - given the number of seconds elapsed, returns a string with the English description of the amount of time passed, consult gen.py in src/utils for the exact argument list and documentation if you feel you could use this function.
- exnToString(e) - improved exception-to-string function. Provides nicer output than a simple str(e).
- InsensitivePreservingDict class - a dict class that is case-insensitive when accessing keys

9.4.4 supybot.utils.iter - iterable utilities

- len(iterable) - returns the length of a given iterable
- groupby(key, iterable) - equivalent to the itertools.groupby function available as of Python 2.4. Provided for backwards compatibility.
- any(p, iterable) - Returns true if any element in the iterable satisfies the predicate p
- all(p, iterable) - Returns true if all elements in the iterable satisfy the predicate p
- choice(iterable) - Returns a random element from the iterable

CHAPTER 10

Using commands.wrap to parse your command's arguments.

This document illustrates how to use the new 'wrap' function present in Supybot 0.80 to handle argument parsing and validation for your plugin's commands.

10.1 Introduction

To plugin developers for older (pre-0.80) versions of Supybot, one of the more annoying aspects of writing commands was handling the arguments that were passed in. In fact, many commands often had to duplicate parsing and verification code, resulting in lots of duplicated code for not a whole lot of action. So, instead of forcing plugin writers to come up with their own ways of cleaning it up, we wrote up the wrap function to handle all of it.

It allows a much simpler and more flexible way of checking things than before and it doesn't require that you know the bot internals to do things like check and see if a user exists, or check if a command name exists and whatnot.

If you are a plugin author this document is absolutely required reading, as it will massively ease the task of writing commands.

10.2 Using Wrap

First off, to get the wrap function, it is recommended (strongly) that you use the following import line:

```
from supybot.commands import *
```

This will allow you to access the wrap command (and it allows you to do it without the commands prefix). Note that this line is added to the imports of plugin templates generated by the supybot-plugin-create script.

Let's write a quickie command that uses wrap to get a feel for how it makes our lives better. Let's write a command that repeats a string of text a given number of times. So you could say "repeat 3 foo" and it would say "foofoofoo". Not a very useful command, but it will serve our purpose just fine. Here's how it would be done without wrap:

```
def repeat(self, irc, msg, args):
    """<num> <text>

    Repeats <text> <num> times.
    """
    (num, text) = privmsg.getArgs(args, required=2)
    try:
        num = int(num)
    except ValueError:
```

```
        raise callbacks.ArgumentError
    irc.reply(num * text)
```

Note that all of the argument validation and parsing takes up 5 of the 6 lines (and you should have seen it before we had privmsg.getArgs!). Now, here's what our command will look like with wrap applied:

```
def repeat(self, irc, msg, args, num, text):
    """<num> <text>

    Repeats <text> <num> times.
    """
    irc.reply(text * num)
repeat = wrap(repeat, ['int', 'text'])
```

Pretty short, eh? With wrap all of the argument parsing and validation is handled for us and we get the arguments we want, formatted how we want them, and converted into whatever types we want them to be - all in one simple function call that is used to wrap the function! So now the code inside each command really deals with how to execute the command and not how to deal with the input.

So, now that you see the benefits of wrap, let's figure out what stuff we have to do to use it.

10.3 Syntax Changes

There are two syntax changes to the old style that are implemented. First, the definition of the command function must be changed. The basic syntax for the new definition is:

```
def commandname(self, irc, msg, args, <arg1>, <arg2>, ...):
```

Where arg1 and arg2 (up through as many as you want) are the variables that will store the parsed arguments. "Now where do these parsed arguments come from?" you ask. Well, that's where the second syntax change comes in. The second syntax change is the actual use of the wrap function itself to decorate our command names. The basic decoration syntax is:

```
commandname = wrap(commandname, [converter1, converter2, ...])
```

Note: This should go on the line immediately following the body of the command's definition, so it can easily be located (and it obviously must go after the command's definition so that commandname is defined).

Each of the converters in the above listing should be one of the converters in commands.py (I will describe each of them in detail later.) The converters are applied in order to the arguments given to the command, generally taking arguments off of the front of the argument list as they go. Note that each of the arguments is actually a string containing the NAME of the converter to use and not a reference to the actual converter itself. This way we can have converters with names like int and not have to worry about polluting the builtin namespace by overriding the builtin int.

As you will find out when you look through the list of converters below, some of the converters actually take arguments. The syntax for supplying them (since we aren't actually calling the converters, but simply specifying them), is to wrap the converter name and args list into a tuple. For example:

```
commandname = wrap(commandname, [(converterWithArgs, arg1, arg2),
                                 converterWithoutArgs1, converterWithoutArgs2])
```

For the most part you won't need to use an argument with the converters you use either because the defaults are satisfactory or because it doesn't even take any.

10.4 Customizing Wrap

Converters alone are a pretty powerful tool, but for even more advanced (yet simpler!) argument handling you may want to use contexts. Contexts describe how the converters are applied to the arguments, while the converters themselves do the actual parsing and validation.

For example, one of the contexts is "optional". By using this context, you're saying that a given argument is not required, and if the supplied converter doesn't find anything it likes, we should use some default. Yet another example is the "reverse" context. This context tells the supplied converter to look at the last argument and work backwards instead of the normal first-to-last way of looking at arguments.

So, that should give you a feel for the role that contexts play. They are not by any means necessary to use wrap. All of the stuff we've done to this point will work as-is. However, contexts let you do some very powerful things in very easy ways, and are a good thing to know how to use.

Now, how do you use them? Well, they are in the global namespace of src/commands.py, so your previous import line will import them all; you can call them just as you call wrap. In fact, the way you use them is you simply call the context function you want to use, with the converter (and its arguments) as arguments. It's quite simple. Here's an example:

```
commandname = wrap(commandname, [optional('int'), many('something')])
```

In this example, our command is looking for an optional integer argument first. Then, after that, any number of arguments which can be anything (as long as they are something, of course).

Do note, however, that the type of the arguments that are returned can be changed if you apply a context to it. So, optional("int") may very well return None as well as something that passes the "int" converter, because after all it's an optional argument and if it is None, that signifies that nothing was there. Also, for another example, many("something") doesn't return the same thing that just "something" would return, but rather a list of "something"s.

10.5 Converter List

Below is a list of all the available converters to use with wrap. If the converter accepts any arguments, they are listed after it and if they are optional, the default value is shown.

- id, kind="integer"
  - Returns something that looks like an integer ID number. Takes an optional "kind" argument for you to state what kind of ID you are looking for, though this doesn't affect the integrity-checking. Basically requires that the argument be an integer, does no other integrity-checking, and provides a nice error message with the kind in it.
- ip
  - Checks and makes sure the argument looks like a valid IP and then returns it.
- int, type="integer", p=None
  - Gets an integer. The "type" text can be used to customize the error message received when the argument is not an integer. "p" is an optional predicate to test the integer with. If p(i) fails (where i is the integer arg parsed out of the argument string), the arg will not be accepted.
- index
  - Basically ("int", "index"), but with a twist. This will take a 1-based index and turn it into a 0-based index (which is more useful in code). It doesn't transform 0, and it maintains negative indices as is (note that it does allow them!).
- color

  - Accepts arguments that describe a text color code (e.g., “black”, “light blue”) and returns the mIRC color code for that color. (Note that many other IRC clients support the mIRC color code scheme, not just mIRC)

- now
  - Simply returns the current timestamp as an arg, does not reference or modify the argument list.
- url
  - Checks for a valid URL.
- httpUrl
  - Checks for a valid HTTP URL.
- email
  - Checks for a syntactically valid email address.
- long, type=”long”
  - Basically the same as int minus the predicate, except that it converts the argument to a long integer regardless of the size of the int.
- float, type=”floating point number”
  - Basically the same as int minus the predicate, except that it converts the argument to a float.
- nonInt, type=”non-integer value”
  - Accepts everything but integers, and returns them unchanged. The “type” value, as always, can be used to customize the error message that is displayed upon failure.
- positiveInt
  - Accepts only positive integers.
- nonNegativeInt
  - Accepts only non-negative integers.
- letter
  - Looks for a single letter. (Technically, it looks for any one-element sequence).
- haveOp, action=”do that”
  - Simply requires that the bot have ops in the channel that the command is called in. The action parameter completes the error message: “I need to be opped to ...”.
- expiry
  - Takes a number of seconds and adds it to the current time to create an expiration timestamp.
- literal, literals, errmsg=None
  - Takes a required sequence or string (literals) and any argument that uniquely matches the starting substring of one of the literals is transformed into the full literal. For example, with `("literal", ("bar", "baz", "qux"))`, you’d get “bar” for “bar”, “baz” for “baz”, and “qux” for any of “q”, “qu”, or “qux”. “b” and “ba” would raise errors because they don’t uniquely identify one of the literals in the list. You can override errmsg to provide a specific (full) error message, otherwise the default argument error message is displayed.
- to
  - Returns the string “to” if the arg is any form of “to” (case-insensitive).
- nick

  - Checks that the arg is a valid nick on the current IRC server.

- seenNick
  - Checks that the arg is a nick that the bot has seen (NOTE: this is limited by the size of the history buffer that the bot has).
- channel
  - Gets a channel to use the command in. If the channel isn't supplied, uses the channel the message was sent in. If using a different channel, does sanity-checking to make sure the channel exists on the current IRC network.
- inChannel
  - Requires that the command be called from within any channel that the bot is currently in or with one of those channels used as an argument to the command.
- onlyInChannel
  - Requires that the command be called from within any channel that the bot is currently in.
- nickInChannel
  - Requires that the argument be a nick that is in the current channel, and returns that nick.
- networkIrc, errorIfNoMatch=False
  - Returns the IRC object of the specified IRC network. If one isn't specified, the IRC object of the IRC network the command was called on is returned.
- callerInGivenChannel
  - Takes the given argument as a channel and makes sure that the caller is in that channel.
- plugin, require=True
  - Returns the plugin specified by the arg or None. If require is True, an error is raised if the plugin cannot be retrieved.
- boolean
  - Converts the text string to a boolean value. Acceptable true values are: "1", "true", "on", "enable", or "enabled" (case-insensitive). Acceptable false values are: "0", false", "off", "disable", or "disabled" (case-insensitive).
- lowered
  - Returns the argument lowered (NOTE: it is lowered according to IRC conventions, which does strange mapping with some punctuation characters).
- anything
  - Returns anything as is.
- something, errorMsg=None, p=None
  - Takes anything but the empty string. errorMsg can be used to customize the error message. p is any predicate function that can be used to test the validity of the input.
- filename
  - Used to get a filename argument.
- commandName
  - Returns the canonical command name version of the given string (ie, the string is lowercased and dashes and underscores are removed).

- text
  - Takes the rest of the arguments as one big string. Note that this differs from the "anything" context in that it clobbers the arg string when it's done. Using any converters after this is most likely incorrect.
- glob
  - Gets a glob string. Basically, if there are no wildcards (`*`, `?`) in the argument, returns `*string*`, making a glob string that matches anything containing the given argument.
- somethingWithoutSpaces
  - Same as something, only with the exception of disallowing spaces of course.
- capability
  - Used to retrieve an argument that describes a capability.
- channelDb
  - Sets the channel appropriately in order to get to the databases for that channel (handles whether or not a given channel uses channel-specific databases and whatnot).
- hostmask
  - Returns the hostmask of any provided nick or hostmask argument.
- banmask
  - Returns a generic banmask of the provided nick or hostmask argument.
- user
  - Requires that the caller be a registered user.
- matches, regexp, errmsg
  - Searches the args with the given regexp and returns the matches. If no match is found, errmsg is given.
- public
  - Requires that the command be sent in a channel instead of a private message.
- private
  - Requires that the command be sent in a private message instead of a channel.
- otherUser
  - Returns the user specified by the username or hostmask in the argument.
- regexpMatcher
  - Gets a matching regexp argument (m// or //).
- validChannel
  - Gets a channel argument once it makes sure it's a valid channel.
- regexpReplacer
  - Gets a replacing regexp argument (s//).
- owner
  - Requires that the command caller has the "owner" capability.
- admin
  - Requires that the command caller has the "admin" capability.

- checkCapability, capability
  - Checks to make sure that the caller has the specified capability.
- checkChannelCapability, capability
  - Checks to make sure that the caller has the specified capability on the channel the command is called in.
- op
  - Checks whether the user has the op mode (+o) set.
- halfop
  - Checks whether the user has the halfop mode (+h) set.
- voice
  - Checks whether the user has the voice mode (+v) set.

10.6 Contexts List

What contexts are available for me to use?

The list of available contexts is below. Unless specified otherwise, it can be assumed that the type returned by the context itself matches the type of the converter it is applied to.

any Looks for any number of arguments matching the supplied converter. Will return a sequence of converted arguments or None.

many Looks for multiple arguments matching the supplied converter. Expects at least one to work, otherwise it will fail. Will return the sequence of converted arguments.

optional Look for an argument that satisfies the supplied converter, but if it's not the type I'm expecting or there are no arguments for us to check, then use the default value. Will return the converted argument as is or None.

additional Look for an argument that satisfies the supplied converter, making sure that it's the right type. If there aren't any arguments to check, then use the default value. Will return the converted argument as is or None.

rest Treat the rest of the arguments as one big string, and then convert. If the conversion is unsuccessful, restores the arguments.

getopts Handles –option style arguments. Each option should be a key in a dictionary that maps to the name of the converter that is to be used on that argument. To make the option take no argument, use "" as the converter name in the dictionary. For no conversion, use None as the converter name in the dictionary.

first Tries each of the supplied converters in order and returns the result of the first successfully applied converter.

reverse Reverse the argument list, apply the converters, and then reverse the argument list back.

commalist Looks for a comma separated list of arguments that match the supplied converter. Returns a list of the successfully converted arguments. If any of the arguments fail, this whole context fails.

10.7 Final Word

Now that you know how to use wrap, and you have a list of converters and contexts you can use, your task of writing clean, simple, and safe plugin code should become much easier. Enjoy!

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

P

Python Enhancement Proposals
    PEP 8, 39