Guacamole Documentation

Release 0.9.2

Zygmunt Krynicki

August 06, 2015

Contents

Contents:

CHAPTER 1

Guacamole - Framework for Creating Command Line Applications

1.1 Tools, done right

Guacamole is a LGPLv3 licensed toolkit for creating good command line applications. Guacamole that does the right things for you and makes writing applications easier.

```
>>> class HelloWorld(guacamole.Command):
...     """A simple hello-world application."""
...     def register_arguments(self, parser):
...         parser.add_argument('name')
...     def invoked(self, ctx):
...         print("Hello {0}!".format(ctx.args.name))
```

Running it directly is as simple as calling `main()`:

```
>>> HelloWorld().main(['Guacamole'], exit=False)
Hello Guacamole!
0
```

What you didn't have to do is what matters:

- configure the argument parser
- define and setup application logging
- initialize internationalization features
- add debugging facilities
- write a custom crash handler

1.2 Features

- Free software: LGPLv3 license
- Documentation: https://guacamole.readthedocs.org.
- Create command classes and run them from command line.
- Group commands to create complex tools.
- Use recipes, ingredients and spices to customize behavior

CHAPTER 2

Installation

The recommended installation method varies per platform. In general `pip`-based installs work everywhere but it is recommended to use other methods if possible.

2.1 Linux Distributions

2.1.1 Debian (and derivatives)

Install either `python-guacamole` or `python3-guacamole` (preferred) using your preferred package manager front-end. An off-line copy of the documentation is available in the `python-guacamole-doc` package. The same package includes all of the bundled examples.

Note: The version of Guacamole available in Debian might not be the most recent version but it was manually reviewed by Debian maintainers. The Debian archive contains cryptographically strong integrity and security guarantees. This method of installation is more trustworthy (and harder to attack) than the one used by pip.

2.1.2 Fedora (and derivatives)

Currently there is no version of Guacamole packaged and available for Fedora. A *copr* repository might be created if there is demand. Proper integration into the Fedora archive is on the roadmap but was not attempted at this time.

2.1.3 Other distributions

There are no other packages as of this writing. Please contribute one if you can. See the *Contributing* for details.

2.2 Other platforms

At the command line run:

```
$ pip install guacamole
```

Note: This section applies to all versions of Windows and OS X.

CHAPTER 3

Usage

Warning: Documentation is under construction. For now please refer to the `examples/` directory in the source distribution.

There are two layers that you might be interested in.

You can either use the existing recipes, most notably, the Command recipe. This is the best way to get started with Guacamole and get advantage of the code that is here already.

The second layer is useful once you start to feel comfortable with the code and want to get more features or perhaps convert your custom code over to work with Guacamole.

Both layers are documented below but first read the philosophy statement.

3.1 Philosophy Statement

The power of Guacamole is based on the simplicity of conventions and sane defaults. Let's talk about some of the conventions that are followed here.

Note: You will see how the philosophy turns into practice in the command turtorial section.

3.1.1 Defaults Matter

Important things make nice applications and tools behave better than random, ad-hoc scripts that have no consistency and happily crash on anything unexpected. Guacamole strives to enable important things that make using applications pleasant.

By default Guacamole will:

- Expose detailed help and usage messages.
- Use translated messages for everything it does.
- Handle logging for you so that it is useful.
- Handle crashes for you so that users can send feedback.
- Use the right directories in your filesystem.
- Use color-coded information, if supported, for readability.
- Teach you, the developer, if you make a mistake that it can detect.

Some defaults say to turn a feature off. Guacamole uses *spices* to let developers opt-into those features that they wish to use. You will learn about spices later in this document. For now just remember that they are equivalent to feature flags.

3.1.2 Documentation Is Important

Documentation is the most important thing you can get wrong easily. You can create perfect tools that do some operation correctly and efficiently but it will all go to waste if nobody can use your product.

Guacamole encourages developers to write useful documentation. The most basic form of documentation is the *docstring*. The docstring is powerful. You see it while writing your code. Other people can see it by various means, using tools like `pydoc` or by reading a document generated with a tool like sphinx.

Guacamole has rich support for documentation. By default, a lot of information is extracted from your command docstrings. You can reuse all of that, for free, to create proper manual pages. Quality tools come with documentation and command line tools use manual pages as the most common, most discoverable means of learning about a particular program.

3.1.3 Internationalization is Important

Internationalization is important to many users. While many developers and system administrators are comfortable with reading English it is strongly recommended to support localization. Modern software gets this right.

Guacamole supports internationalization by default. Commands can advertise their gettext domain using the `gettext_domain` attribute (see *`get_gettext_domain()`* for details). Guacamole will carefully work with your docstrings to feed them to gettext and extract the useful bits out.

Commands can mix-and-match different gettext domains without issues. If you are writing a non-trivial application which is composed of commands coming from various sources they will all work correctly together.

3.1.4 Convention over Configuration

Guacamole has a lot of APIs. Most of the time you won't have to work with them. Guacamole will reuse information that you can provide without defining methods.

This is how the docstrings are used for documentation. This is how you can define numerous attributes to describe specific features of your commands. Instead of working with the methods you can just define an item. This has the advantage that Guacamole can look at your command class and can educate you if you make a mistake. This is easier to work with than reading through back-traces or working with type annotations that may or may not be enough to capture something you want to express.

3.2 Recipes, Ingredients and Spices

Guacamole is a framework for creating command line applications. To understand how to use it, you need to know about the three concepts (recipes, ingredients and spices). They define how guacamole works (tastes) and they are how you can make guacamole work for you in new and interesting ways.

3.2.1 Ingredients

Ingredients are pluggable components that can be added to a guacamole recipe. They have well-defined APIs and are invoked by guacamole during the lifetime of the application. You can think of ingredients as of middleware or a

fancy context manager. For an in-depth documentation see the *Ingredient* class. For a list of bundled ingredients (batteries included) please see *bundled-ingredients*.

Guacamole uses ingredients to avoid having complex, convoluted core. The core literally does nothing more than to invoke all ingredients in a given order. Applications use ingredietns indirectly, through recipes.

3.2.2 Spices

Spices are small, optional bits of taste that can be added along with a given ingredient. They are just a feature flag with a fancy name. You will see spices documented along with each ingredient. For many features you will use the sane defaults that guacamole aims to provide but sometimes you may want to tweak something. Such elements can be hidden behind an ingredient.

Guacamole uses spices to offer fixed cusomizability where it makes sense to do so. Applications say witch spices they wish to use. Spices always enable non-default behavior.

3.2.3 Recipes

Recipes define the sequence of ingredients to use for a tasty guacamole. In reality a recipe is a simple function that returns a list of ingredient instances to use in a given application.

Guacamole uses recipes to offer easy-to-use, well-designed patterns for creating applications. Anyone can create a recipe that uses a set of ingredients that fit a particular purpose.

3.2.4 Command?

The *Command* class is just a recipe that uses a set of ingredients. As Guacamole matures, other recipes may be added.

3.3 Using Stock Recipes

3.3.1 The Command Recipe

The command recipe contains the distilled, correct behavior for command line applications. The main face of the command recipe is the *Command* class.

Note: Guacamole values conventions. Instead of overriding many of the methods that comprise the Command class, you can just define a variable that will take priority. This leads to shorter and more readable code.

Defining commands

Let's build a simple hello-world example again:

```
>>> class HelloWorld(guacamole.Command):
...     def invoked(self, ctx):
...         print("Hello World!")
```

The central entry point of each command is the *invoked()* method. The method is called once the command is ready to be dispatched. This is what you would put inside your `main()` function, after the boiler-plate code that Guacamole handles for you. What you do here is up to you.

For now, let's just run our simple example with the convenience method *main()*. Note that here we're passing extra arguments to control how the tool executes, normally you would just call main without any arguments and it will do the right thing.

```
>>> HelloWorld().main([], exit=False)
Hello World!
0
```

For now let's ignore the argument *ctx*. It is extremely handy, as we will see shortly, but we don't need it yet.

Note: This little example is available in the `examples/` directory in the source distribution. The version of Guacaomle packaged in Debian has them in the directory `/usr/share/doc/python-guacamole-doc/examples`. As the directory name implies, you have to install the `python-guacamole-doc` package to get them.

Do use the example and play around with it, see how it behaves if you run it with various arguments. The idea is that Guacamole is supposed to create *good* command line applications. Good applications do the right stuff internally. The `hello-world` example is trivial but we'll see more of what is going on internally soon.

Working with arguments & The Context

Commands typically take arguments. To say which arguments are understood by our command we need to implement the second method *register_arguments()*. This method is called with the familiar `argparse.ArgumentParser` instance. You've seen this code over and over, here you should just focus on configuring the arguments and options. Guacamole handles the parser for you.

```
>>> class HelloWorld(guacamole.Command):
...     def register_arguments(self, parser):
...         parser.add_argument('name')
...     def invoked(self, ctx):
...         print("Hello {0}!".format(ctx.args.name))
```

As you can see, the context is how you reach the command line arguments parsed by *argparse*. What else is there you might ask? The answer is *everything*.

The context is how *ingredients* can expose useful capabilities to commands. The command recipe is comprised of several ingredients, as you will later see. One of those ingredients parsers command line arguments and adds the results to the context as the `args` object.

Note: When reading documentation about particular ingredients make sure to see how they interact with the context. Each ingredient documents that clearly.

Let's run our improved command and see what happens:

```
>>> HelloWorld().main(["Guacamole"], exit=False)
Hello Guacamole!
0
```

No surprises there. We can see that the command printed the hello message and then returned the exit code `0`. The exit code is normally passed to the system so that your application can be scripted.

Note: Guacamole will return `0` for you if you don't return anything. If you do return a value we'll just preserve it for you. You can also raise SystemExit with any value and we'll do the right thing yet again.

This should be all quite familiar to everyone so we won't spend more time on arguments now. You can read the argparse-tutorial if you want.

A small digression, why argparse?

By default, all command line parsing is handled by `argparse`.

Guacamole doesn't force you to use argparse (nothing really is wired to depend on it in the core) but the stock set of ingredients do use it. Argparse is familiar to many developers and by having it by default you can quickly convert your application code over to guacamole without learning two new things at a time.

Nesting Commands

Many common tools expose everything from a top-level command, e.g. `git commit`. Here, `git` gets invoked, looks at the command line arguments and delegates the dispatching to the `git-commit` command.

All Guacamole commands can be nested. Let's build a quick git-like command to see how to do that.

```
>>> class git_commit(guacamole.Command):
...     name = 'commit'
...     def invoked(self, ctx):
...         print("commit invoked")

>>> class git_log(guacamole.Command):
...     def invoked(self, ctx):
...         print("log invoked")

>>> class git(guacamole.Command):
...     name = 'git'
...     sub_commands = (
...         (None, git_commit),
...         ('log', git_log),
...     )
```

As you see it's all based on declarations. Each command now cares about the name it is using. Names can be assigned in the `sub_commands` list or individually in each class, by defining the `name` attribute.

The name listed in sub_commands takes precedence over the name defined in the class. Here, the `git_log` command doesn't define a `name` so we provide one explicitly as the first element of the pair, as sequence of which is stored in `sub_commands`.

Note: Behind the scenes Guacamole actually calls a number of methods for everything. See *`get_sub_commands()`* and *`get_cmd_name()`* for the two used here. There are *many* more methods though.

Let's invoke our fake git to see how that works now:

```
>>> git().main(["commit"], exit=False)
commit invoked
0

>>> git().main(["log"], exit=False)
log invoked
0
```

So far everything behaves as expected. Let's see what happens if we run something that we've not coded:

```
>>> git().main(["status"], exit=False)
2
```

This won't fit the *doctest* above (it's printed on stderr) but in reality the application will also say something like this:

```
usage: git [-h] {commit,log} ...
setup.py: error: invalid choice: 'status' (choose from 'commit', 'log')
```

Note: Technically the *Command* class has numerous methods. Most of those methods are of no interest to most of the developers. Feel free to read the API reference later if you are interested.

CHAPTER 4

Ingredients

This section contains documentation for each of the available ingredients.

4.1 ANSI

4.1.1 Summary

Ingredient for working with ANSI Control Codes

4.1.2 Description

The ANSI ingredient exposes ANSI control codes in a simple way. ANSI codes can be used to control text color and style on compatible terminals.

Linux terminal emulators commonly support a wide subset of control codes. Particular support differs between the classic Linux console, *Xterm*, *gnome-terminal* and *konsole* (and the backing libraries). Some features are supported more widely than others. In particular, the text console is rather limited and will likely remain so until the systemd-based replacement is commonly used.

The terminal emulator included in Apple's OS X supports a subset of the features (3rd party terminal emulators for OS X were not tested, contributions are welcome). In general you can treat OS X like a poor version of Linux.

The windows command prompt is the most limited environment as it only support several foreground and background colors and nothing else at all. It also has issues with Unicode (as in, it doesn't support it at all). On Windows, usage of ANSI depends on the availability of `colorama`. Colorama is a third party library that wraps `sys.stdout` and `sys.stderr`, parses ANSI control codes and converts them to the corresponding Windows API calls.

4.1.3 Spices

This ingredient is not influenced by any *spices*.

4.1.4 Context

This ingredient adds two objects to the context:

`ansi` An instance of *`ANSIFormatter`*. The object is automatically configured (disabled) when the extra control codes are undesired (stdout not attached to a terminal emulator).

aprint The *aprint()* method, as a shorthand for `ctx.ansi.aprint`.

4.1.5 Command Line Arguments

This ingredient is not exposing any command line arguments.

4.1.6 Examples

Let's construct a simple example. Note that typically you will use the context that is provided to you from the *invoked()* method of a command.

```
>>> from guacamole.core import Context
>>> from guacamole.ingredients import ansi
>>> ctx = Context()
>>> ansi.ANSIIngredient(enable=True).added(ctx)
```

The context now has the `ansi` object, which is an instance of *ANSIFormatter*.

It has some methods and properties that we'll see below but it is also callable and darn convenient to use.

You can use the `fg` and `bg` keyword arguments to control the *foreground* and *background* text color respectively.

```
>>> str(ctx.ansi('red on blue', fg='red', bg='blue'))
'\x1b[31;44mred on blue\x1b[0m'
```

You can use keyword arguments that correspond to *each* of the countless `sgr_` constants available in the class *ANSI*. Here, let's get bold text using the `sgr_bold` code.

```
>>> str(ctx.ansi('bold text', bold=1))
'\x1b[1mbold text\x1b[0m'
```

In some cases you may want to use different code knowing that the output will be colorized (e.g. use color codes instead of longer text labels). You can achieve that by testing :meth'~guacamole.ingredients.ansi.ANSI.is_enabled'.

```
>>> # Let's disable the ANSI support for this test
>>> ansi.ANSIIngredient(enable=False).added(ctx)
>>> if ctx.ansi.is_enabled:
...     ctx.aprint('!!!!', fg='red')
... else:
...     ctx.aprint('ALARM')
ALARM
```

4.1.7 Expressing colors

Guacaomle supports several styles of colors:

- Named colors represented as strings:
  - `"black"`
  - `"red"`
  - `"green"`
  - `"yellow"`
  - `"blue"`
  - `"magenta"`

  - `"cyan"`
  - `"white"`
- Bright variant of named colors (not repeated)
- Indexed colors represented as an integer in range(256):
  - 0x00-0x07: standard colors (as in `ESC [ 30-37 m`)
  - 0x08-0x0F: high intensity colors (as in `ESC [ 90-97 m`)
  - 0x10-0xE7: 6 × 6 × 6 = 216 colors: 16 + 36 × r + 6 × g + b (0 ≤ r, g, b ≤ 5)
  - 0xE8-0xFF: grayscale from black to white in 24 steps
- RGB colors represented as (r, g, b) where each component is an integer in range(256)
- The special value `"auto"` which picks the complementary (readable) variant. Auto may be used in one of `fg=` or `bg=` if `bg=` or `fg=` (respectively) are using a concrete color.

Note: The actual colors behind the string-named colors vary between different terminal emulators. Sometimes the color is just slightly different. Sometimes it is just totally unrelated to the one specified in the ANSI standard.

Warning: RGB colors are not supported on Windows and OS X. They are only supported on modern terminal emulators, typically on Linux distributions.

4.2 CommandTreeBuilder

4.2.1 Summary

Ingredient for arranging all the *Command* classes into a tree of objects.

4.2.2 Description

The command tree builder ingredient is a part of the command recipe. It is responsible for instantiating all sub-commands and arranging them into a tree for other ingredients to work with (most notably the argument parser ingredient).

The secondary task is to add the *spices* requested by the top-level command to the bowl. This lets other ingredients act differently and effectively allows the top-level command to influence the runtime behavior of the whole recipe.

4.2.3 Spices

This ingredient is not influenced by any *spices*.

4.2.4 Context

This ingredient adds two objects to the context:

`cmd_tree` A tree of tuples that describes all of the commands and their sub commands.

`cmd_toplevel` The top-level command object.

In addition, this ingredient inspects the *spieces* required by the top-level command and adds them to the bowl.

4.2.5 Command Line Arguments

This ingredient is not exposing any command line arguments.

4.2.6 Examples

Let's create two examples below. One for a simple command and another for a hierarchical command. This example will not use the full command recipe, to focus on the side effects of just the command tree builder ingredient.

Flat Command

We'll need a command object:

```
>>> from guacamole.recipes.cmd import Command
>>> class HelloWorld(Command):
...     pass
```

Note that the tree builder is called with an *instance* of the command, not the class. This allows the top-level command to have a custom initializer, which might be helpful.

```
>>> from guacamole.core import Context
>>> from guacamole.ingredients import cmdtree
>>> ctx = Context()
>>> cmdtree.CommandTreeBuilder(HelloWorld()).added(ctx)
```

The context now has the `cmd_toplevel` object which is just the instance of the command we've used.

```
>>> ctx.cmd_toplevel
<HelloWorld>
```

Similarly, we'll have a tree of all the commands and their names in `cmd_tree`:

```
>>> ctx.cmd_tree
cmd_tree_node(cmd_name=None, cmd_obj=<HelloWorld>, children=())
```

The first element of the tuple is the effective command name. This can be used to rename a sub-command. Note that typically the `command.name` attribute is used (see *`get_cmd_name()`*). The second element is the instance and the last element is a tuple of identical `cmd_tree_node` tuples, one for each of the sub-commands. We'll see how that looks like in the next example.

Nested Commands

We'll need a few commands for this example. Let's replicate the `git`, `git commit`, `git stash`, `git stash pop` and `git stash list` commands.

```
>>> from guacamole.recipes.cmd import Command
>>> class StashList(Command):
...     pass
>>> class StashPop(Command):
...     pass
>>> class Stash(Command):
...     sub_commands = (('list', StashList), ('pop', StashPop))
>>> class Commit(Command):
...     pass
>>> class Git(Command):
...     sub_commands = (('commit', Commit), ('stash', Stash))
```

Now, let's feed the `Git` class to the context.

```
>>> from guacamole.core import Context
>>> from guacamole.ingredients import cmdtree
>>> ctx = Context()
>>> cmdtree.CommandTreeBuilder(Git()).added(ctx)
```

The `cmd_toplevel` is as before (the `Git` *instance*). Let's look at the more interesting command tree.

```
>>> ctx.cmd_tree
cmd_tree_node(cmd_name=None, cmd_obj=<Git>,
    children=(cmd_tree_node(cmd_name='commit', cmd_obj=<Commit>,
    children=()), cmd_tree_node(cmd_name='stash',
    cmd_obj=<Stash>, children=(cmd_tree_node(cmd_name='list',
    cmd_obj=<StashList>, children=()), cmd_tree_node(cmd_name='pop',
    cmd_obj=<StashPop>, children=()))))
```

Blah, that's mouthful. Let's see particular fragments to understand it better.

```
>>> ctx.cmd_tree.children[0].cmd_name
'commit'
>>> ctx.cmd_tree.children[1].cmd_name
'stash'
>>> ctx.cmd_tree.children[1].children[0].cmd_name
'list'
>>> ctx.cmd_tree.children[1].children[1].cmd_name
'pop'
```

Most of the time you won't have to use this data. Typically, it is consumed by the argument parser ingredient. Still, if you need it, here it is.

4.3 CommandTreeDispatcher

4.3.1 Summary

Ingredient for executing the *`invoked()`* methods of all the commands that were selected by the user on command line.

4.3.2 Description

This ingredient is responsible for invoking commands. It works during the dispatch phase of the application life-cycle. Since earlier stages can be interrupted it is not aways reached. E.g. when the application is invoked with the `--help` argument.

The way this ingredient works is simple. It assumes that the argument parser creates a specific structure of references to command objects. The structure is stored in the `argparse` name-space object (which is available in `ctx.args` after the parsing phase. The structure is a sequence of attributes `ctx.args.command0`, `ctx.args.command1`, `ctx.args.command2`, etc. The first one, `ctx.args.command0` is always present. Subsequent attributes are present if sub-commands are specified on the command line. For example, keeping our git sample in mind, the following command:

```
$ git stash
```

Will result in `ctx.args.command0` instance of the *Git* command and `ctx.args.command1` an instance of the *GitStash* command. The dispatcher ingredient will invoke the `command0`, look at the return value and then (most likely) proceed to `command1` (N+1 in general).

The way return value is interpreted is interesting. In general, there are three cases:

- None is interpreted as "nothing special happened". In the example above. The `git stash` will first call `Git.invoked()`, see the (default) None and will proceed to call `GitStash.invoked()`.
- A generator is interpreted as a context-manager like. This allows, for example, the `git` command to use a context manager in its `invoked()` method to provide some managed resource to each sub-command. Note that the *invoked* method must behave as it if was decorated with `@functools.contextmanager` but it must not be actually decorated like that.
- Any other return value is interpreted as an error code and stops recursive command dispatch. It will be finally returned from the `main()` method or raised as a `SystemExit` exception.

4.3.3 Spices

This ingredient is not influenced by any *spices*.

4.3.4 Context

This ingredient does not change the context. It does depend on the `args` object that is published by the argument parser ingredient.

4.3.5 Command Line Arguments

This ingredient is not exposing any command line arguments.

4.3.6 Examples

Let's see how command invocation works in the few specific examples below.

Single Command

Let's start with a hello-world command first:

```
>>> from guacamole.recipes.cmd import Command
>>> class HelloWorld(Command):
...     def invoked(self, ctx):
...         print("Hello World")
```

Let's create the necessary infrastructure for using the dispatcher:

```
>>> import argparse
>>> from guacamole.core import Context
>>> from guacamole.ingredients import cmdtree
>>> ctx = Context()
>>> ctx.args = argparse.Namespace()
```

Now let's run the *HelloWorld* command:

```
>>> ctx.args.command0 = HelloWorld()
>>> cmdtree.CommandTreeDispatcher().dispatch(ctx)
Hello World
```

Success! The print worked and we also got the exit code (None, which is not printed by the repl).

Next, let's implement the classic UNIX `false(1)` command:

```
>>> class false(Command):
...     def invoked(self, ctx):
...         return 1
```

Now, let's invoke it:

```
>>> ctx.args.command0 = false()
>>> cmdtree.CommandTreeDispatcher().dispatch(ctx)
1
```

One. Also good.

All command line tools return an exit code. If you actually run this command in the shell you can inspect the return code in several ways (depending on what is your shell). On Windows that is:

```
echo %ERRORLEVEL%
```

And on all other systems, that are mostly using Bash by default:

```
echo $?
```

In both cases, you should see `1` being printed by those echo statements.

Nested Commands

Let's expand the Git example to examine the context-manager-like behavior.

```
>>> class GitLibrary(object):
...     def __enter__(self):
...         print("Git initialized")
...         return self
...     def __exit__(self, *args):
...         print("Git finalized")
...     def commit(self):
...         print("Using git to commit")

>>> class Commit(Command):
...     def invoked(self, ctx):
...         with GitLibrary() as git:
...             git.commit()

>>> class Git(Command):
...     sub_commands = (('commit', Commit),)
```

Now, let's see what dispatch does here:

```
>>> ctx.args.command0 = Git()
>>> ctx.args.command1 = Commit()
>>> cmdtree.CommandTreeDispatcher().dispatch(ctx)
Git initialized
Using git to commit
Git finalized
```

If you have many commands that need to use some shared resource, you may be tempted to move the initialization to a shared code path. Guacamole allows you to do this by calling **all** the `invoked()` methods of all of the commands specified on command line.

Let's modify the example to show this. The git library code will say as-is. The commit and git commands will be changed, to move the initialization code around.

```
>>> class Commit(Command):
...     def invoked(self, ctx):
...         ctx.git.commit()

>>> class Git(Command):
...     sub_commands = (('commit', Commit),)
...     def invoked(self, ctx):
...         with GitLibrary() as git:
...             ctx.git = git
...             yield
```

Now, let's see what dispatch does now:

```
>>> ctx.args.command0 = Git()
>>> ctx.args.command1 = Commit()
>>> cmdtree.CommandTreeDispatcher().dispatch(ctx)
Git initialized
Using git to commit
Git finalized
```

No change, that's running exactly as before but now we can add more commands without duplicating the relevant code over and over.

Note: Here, the finalization will happen even if something bad happens (e.g. `Commit` raising an exception). It's not useful often but it can be a way to use the context manager protocol with commands.

4.4 VerboseCrashHandler

4.4.1 Summary

Ingredient for handling crashing commands.

4.4.2 Description

This ingredient mimics the default behavior of python for an uncaught exception. That is, to print the exception details, the function backtrace and to exit the process.

4.4.3 Spices

This ingredient is not influenced by any *spices*.

4.4.4 Context

This ingredient does not add any objects to the context. This ingredient does use the context though, to access the crash meta-data. This includes:

`exc_type` The class of the exception that caused the application to crash.

`exc_value` The exception object itself.

traceback The traceback object.

Note: The three attributes are automatically added by the *Bowl* when something bad happens.

4.4.5 Command Line Arguments

This ingredient is not exposing any command line arguments.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/zyga/guacamole/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

5.1.4 Write Documentation

Guacamole could always use more documentation, whether as part of the official Guacamole docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/zyga/guacamole/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *guacamole* for local development.

1. Fork the *guacamole* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/guacamole.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv guacamole
$ cd guacamole/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 guacamole
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.2, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/zyga/guacamole/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest guacamole.test_core
```

(Where guacamole.test_core is the module with tests you want to run)

CHAPTER 6

Credits

6.1 Development Lead

- Zygmunt Krynicki <zygmunt.krynicki@canonical.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.9.2 (2015-08-06)

- Fix https://github.com/zyga/guacamole/issues/11

7.2 0.9.1 (2015-08-06)

- Fix https://github.com/zyga/guacamole/issues/9

7.3 0.9 (2015-05-11)

- Vastly improved documentation
- Bugfixes and changes based on early feedback
- New cmdtree module with two ingredients (for instantiating commands and for dispatching the invoked method)
- Simplified argparse ingredient (for just handling parser)
- Unit tests and doctests for some of the functionality

7.4 0.8 (2015-04-21)

- First release on PyPI.

7.5 2012-2015

- Released on PyPI as a part of plainbox as `plainbox.impl.clitools`, `plainbox.impl.logging`, `plainbox.i18n` and `plainbox.impl.secure.plugins`.

CHAPTER 8

Code Reference

8.1 guacamole.core

The essence of guacamole.

This module defines the three essential core classes: *Ingredient*, *Bowl*, *Context*. All of those have stable APIs.

class guacamole.core.**Ingredient**

Part of guacamole.

Ingredients are a mechanism for inserting functionality into Guacamole. The sequence of calls to ingredient methods is as follows:

- *added()*

The added method is where an ingredient can advertise itself to other ingredients that it explicitly collaborates with.

- *preparse()*

The preparse method is where ingredients can have a peek at the command line arguments. This can serve to optimize further actions. Essentially guacamole allows applications to parse arguments twice and limit the actions needed to do that correctly to the essential minimum required.

- *early_init()*

The early initialization method can be used to do additional initialization. It can take advantage of the fact that the whole command line arguments are now known and may have been analyzed further by the preparse method.

- *parse()*

The parse method is where applications are expected to fully understand command line arguments. This method can abort subsequent execution if arguments are wrong in in some way. After parsing command line arguments the application should be ready for execution.

- *late_init()*

The late initialization method mimics the early initialization method but is called after parsing all of the command line arguments. Again, it can be used to prepare addiotional resources necessary for a given application.

- *dispatch()*

The dispatch method is where applications execute the bulk of their actions. Dispatching is typically done with one of the standard ingredients which will locate the appropriate method to call into the application.

Depending on the outcome of the dispatch (if an exception is raised or not) one of `dispatch_succeeded()`‘() or *dispatch_failed()* is called.

- *shutdown()*

This is the last method called on all ingredients.

Each of those methods is called with a context argument (`Context:`). A context is a free-for-all environment where ingredients can pass data around. There is no name-spacing. Ingredients should advertise what they do with the context and what to expect.

added(*context*)
Ingredient method called before anything else.

build_early_parser(*context*)
Ingredient method called to build the early parser.

build_parser(*context*)
Ingredient method called to build the full parser.

dispatch(*context*)
Ingredient method for dispatching (execution).

Note: The first ingredient that implements this method and returns something other than None will stop command dispatch!

---

dispatch_failed(*context*)
Ingredient method called when dispatching fails.

dispatch_succeeded(*context*)
Ingredient method called when dispatching is correct.

early_init(*context*)
Ingredient method for early initialization.

late_init(*context*)
Ingredient method for late initialization.

parse(*context*)
Ingredient method called to parse command line arguments.

preparse(*context*)
Ingredient method called to pre-parse command line aruments.

shutdown(*context*)
Ingredient method called after all other methods.

class `guacamole.core.`**Context**
Context for making guacamole with ingredients.

A context object is created and maintained throughout the life-cycle of an executing tool. A context is passed as argument to all ingredient methods.

Since context has no fixed API anything can be stored and loaded. Particular ingredients document how they use the context object.

class `guacamole.core.`**Bowl**(*ingredients*)
A vessel for preparing guacamole out of ingredients.

Note: Each Bowl is single-use. If you eat it you need to get another one as this one is dirty and cannot be reused.

---

add_spice(*spice*)
Add a single spice the bowl.

eat(*argv=None*)
Eat the guacamole.

Parameters **argv** – Command line arguments or None. None means that sys.argv is used

Returns Whatever is returned by the first ingredient that agrees to perform the command dispatch.

The eat method is called to run the application, as if it was invoked from command line directly.

has_spice(*spice*)
Check if a given spice is being used.

This method can be used to construct checks if an optional ingredient feature should be enabled or not. Spices are simply strings that describe optional features.

8.2 guacamole.recipes

APIs for guacamole add-on developers.

This module contains the public APIs for add-on developers. Add-ons (or plug-ins) for guacamole are called **ingredients**. The `Ingredient` class contains a description of available add-on methods.

Ingredients are somewhat similar to Django middleware as they can influence the execution of an application across its life-cycle. All of core guacamole features are implemented as ingredients. Developers are encouraged to read core ingredients to understand how to formulate their own design.

Ingredient APIs are *public*. They will be maintained for backwards compatibility. Since Guacamole doesn't automatically enable any third-party ingredients, application developers that wish to use them need to use the *guacamole.core* module to create their own guacamole out of available ingredients. Ingredient developers are recommended in documenting how to use each ingredient this way.

In addition this module contains the public APIs for creating custom mixes of guacamole. A custom mix begins with a *Bowl* with any number of *Ingredient* objects added.

If you are familiar with the *Command* class you should know that they are using the recipe system internally. They refer to pre-made recipes that put particular ingredients into the bowl for a ready dish.

If you wish to build a custom experience on top of guacamole, please provide a new recipe class. Recipes are how applications should interact with any guacamole mixtures.

class guacamole.recipes.**Recipe**
Mechanism to use ingredients to dispatch and invoke commands.

get_ingredients()
Get a list of ingredients for making guacamole.

Returns A list of initialized ingredients.

Raises RecipeError If the recipe is wrong. This is a developer error. Do not handle this exception. Consult the error message to understand what the problem is and correct the recipe instead.

main(*argv=None*, *exit=True*)
Shortcut to prepare a bowl of guacamole and eat it.

Parameters

- **argv** – Command line arguments or None. None means that sys.argv is used

- **exit** – Raise SystemExit after finishing execution

Returns Whatever is returned by the eating the guacamole.

Raises Whatever is raised by eating the guacamole.

Note: This method always either raises and exception or returns an object. The way it behaves depends on the value of the *exit* argument.

This method can be used to quickly take a recipe, prepare the guacamole and eat it. It is named main as it is applicable as the main method of an application.

The *exit* argument controls if main returns normally or raises SystemExit. By default it will raise SystemExit (it will either wrap the return value with SystemExit or re-raise the SystemExit exception again). If SystemExit is raised but *exit* is False the argument to SystemExit is unwrapped and returned instead.

prepare()
Prepare a bowl with the ingredients specified by this recipe.

Returns A new `Bowl` instance with all the ingredients prepared.

exception `guacamole.recipes.`**RecipeError**
Exception raised when the recipe for guacamole is incorrect.

This exception is only used when a set of ingredients is ordered correctly or has some missing elements. Each time this exception is raised it is accompanied by a detailed message that should help you to resolve the problem.

Note: This exception should not be handled, it is a developer error.

8.3 guacamole.recipes.cmd

Recipe for using guacamole to run commands.

This module contains sock recipes for guacamole. Stock recipes allow application developers to use use simple-to-understand design patterns to get predictable runtime behiavior.

Currently, guacamole ships with two such recipes, for creating simple commands and for creating hierarhical command groups. They are captured by the *Command* and `Group` classes respectively.

class `guacamole.recipes.cmd.`**Command**
A single-purpose command.

Single purpose commands are the most commonly known tools in command line environments. Tools such as `ls`, `mkdir` or `vim` all fall in this class. A command is essentially a named action that can be invoked from the terminal emulator or other command line environment specific to a given operating system.

To create a new command simply create a custom class and override the *invoked()* method. Put all of your custom code there. If you want to interact with command line arguments then please also override the *register_arguments()* method.

Have a look at example applications for details of how to do this. You can use them as a starting point for your own application as they are licensed very liberally.

get_app_id()
Get the identifier of the application.

Note: Application identifier is looked up using the `app_id` attribute.

The syntax of a valid command identifier is `REVERSE-DNS-NAME:ID`. For example, `"com.example.product:command"`. This identifier must not contain characters that are hostile to the file systems. It's best to stick to ASCII characters and digits.

On *Mac OS X* this will be used as a directory name rooted in `~/Library/Preferences/`. On Linux and other freedesktop.org-based systems this will be used as directory name rooted in `$XDG_CONFIG_HOME` and `$XDG_CACHE_HOME`. On Windows it will be used as a directory name rooted in the per-user `AppData` folder.

Note: If this method returns None then logging and configuration services are disabled. It is strongly recommended to implement this method and return a correct value as it enhances application behavior.

---

get_app_name()
Get the name of the application.

Note: Application name is looked up using the `app_name` attribute.

Application name differs from executable name. The executable might be called `my-app` or `myapp` while the application might be called `My Application`.

get_app_vendor()
Get the name of the application vendor.

The name should be a human readable name, like `"Joe Developer"` or `"Big Corporation Ltd."`

Note: Application vendor name is looked up using the `app_vendor` attribute.

---

get_cmd_description()
Get the leading, multi-line description of this command.

Returns `self.description`, if defined

Returns A substring of the class docstring between the first line (which is discarded) and the string `@EPILOG@`, if present, or the end of the docstring, if any

Returns None, otherwise

The description string will be displayed after the usage string but before any of the detailed argument descriptions.

Please consider following good practice by keeping the description line short enough not to require scrolling but useful enough to provide additional information that cannot be inferred from the name of the command or other arguments. Stating the purpose of the command is highly recommended.

get_cmd_epilog()
Get the trailing, multi-line description of this command.

Returns `self.epilog`, if defined

Returns A substring of the class docstring between the string `@EPILOG` and the end of the docstring, if defined

Returns None, otherwise

The epilog is similar to the description string but it is instead printed after the section containing detailed descriptions of all of the command line arguments.

Please consider following good practice by providing additional details about how the command can be used, perhaps an example or a reference to means of finding additional documentation.

get_cmd_help()
Get the single-line help of this command.

Returns `self.help`, if defined

Returns The first line of the docstring, without the trailing dot, if present.

Returns None, otherwise

get_cmd_name()
Get the name of the application executable.

Note: If this method returns None then the executable name is guessed from `sys.argv[0]`.

get_cmd_spices()
Get a list of spices requested by this command.

Feature flags are a mechanism that allows application developers to control ingredients (switch them on or off) as well as to control how some ingredients behave.

Returns `self.spices`, if defined. This should be a set of strings. Each string represents as single flag. Ingredients should document the set of flags they understand and use.

Returns An empty set otherwise

Some flags have a generic meaning, you can scope a flag to a given ingredient using the `name:` prefix where the name is the name of the ingredient.

get_cmd_usage()
Get the usage string associated with this command.

Returns `self.usage`, if defined

Returns None, otherwise

The usage string typically contains the list of available, abbreviated options, mandatory arguments and other arguments. Its purpose is to quickly inform the user on the basic syntax used by the command.

It is perfectly fine not to customize this method as the default is to compute an appropriate usage string out of all the arguments. Consider implementing this method in a customized way if your command has highly complicated syntax and you want to provide an alternative, more terse usage string instead.

get_cmd_version()
Get the version reported by this executable.

Note: If this method returns None then the `--version` option is disabled.

get_gettext_domain()
Get the gettext translation domain associated with this command.

The value returned will be used to select translations to global calls to gettext() and ngettext() everywhere in python.

Note: If this method returns None then all i18n services are disabled.

get_locale_dir()
Get the path of the gettext translation catalogs for this command.

This value is used to bind the domain returned by *get_gettext_domain()* to a specific directory.

Note: If this method returns None then standard, system-wide locations are used (on compatibles systems). In practical terms, on Windows, you may need to use it to have access to localization data.

get_sub_commands()
Get a list of sub-commands of this command.

Returns `self.sub_commands`, if defined. This is a sequence of pairs `(name, cls)` where `name` is the name of the sub command and `cls` is a command class (not an object). The `name` can be None if the command has a version of *get_cmd_name()* that returns an useful value.

Returns An empty tuple otherwise

Applications can create hierarchical commands by defining the `sub_commands` attribute. Many developers are familiar with nested commands, for example `git commit` is a sub-command of the `git` command. All commands can be nested this way.

invoked(*context*)
Callback called when the command gets invoked.

Parameters **context** – The guacamole context object.

Returns The return value is returned by the executable. It should be an integer between 0 and 255. Other values are will likely won't work at all.

The context argument can be used to access command line arguments and other information that guacamole provides.

main(*argv=None*, *exit=True*)
Shortcut for running a command.

See *guacamole.recipes.Recipe.main()* for details.

register_arguments(*parser*)
Callback called to register command-specific arguments.

Parameters **parser** – Argument parser (from `argparse`) specific to this command.

class guacamole.recipes.cmd.**CommandRecipe**(*command*)
A recipe for using commands.

get_ingredients()
Get a list of ingredients for guacamole.

8.4 guacamole.ingredients

Package with ingredients bundled with guacamole.

8.5 guacamole.ingredients.cmdtree

Ingredient for arranging commands into a tree structure.

class guacamole.ingredients.cmdtree.**CommandTreeBuilder**(*command*)
Ingredient for arranging commands into a tree of instances.

Since commands and sub-commands are specified as classes there has to be an ingredient that instantiates them and resolves all the naming ambiguities. Here it is.

This component acts early, in its *added()* method.

added(*context*)
Ingredient method called before anything else.

Here this method just builds the full command tree and stores it inside the context as the `cmd_tree` attribute. The structure of the tree is explained by the `build_cmd_tree()` function.

class guacamole.ingredients.cmdtree.**CommandTreeDispatcher**
Ingredient for dispatching commands hierarchically.

This ingredient builds on the *CommandTreeBuilder* ingredient. It implements the *dispatch()* method that recurses from the top (root) of the command tree down to the appropriate leaf, calling the invoke() method of each command.

The process stops on the first command that returns a value other than None, raises an exception or until a leaf command is reached. THe ability to return early allows commands to perform some sanity checks or short-circuit execution that is hard to express using standard parser APIs.

Lastly, a command can return a generator, this is treated as a sign that the generator implements a context-manager-like API. In this case the generator is called exactly twice and can be used to manage resources during the lifetime of all sub-commands.

dispatch(*context*)
Dispatch execution to the invoke() method of selected commands.

class guacamole.ingredients.cmdtree.**cmd_tree_node**(*cmd_name*, *cmd_obj*, *children*)
A named tuple for representing the hierarchy of commands.

children
Alias for field number 2

cmd_name
Alias for field number 0

cmd_obj
Alias for field number 1

8.6 guacamole.ingredients.argparse

Ingredients for using arparse for parsing command line arguments.

This module contains two ingredients. The main one is the *ParserIngredient*. It is responsible for handling all of the command line parsing and command argument registration. It is a part of the recipe for the command class. Note that command dispatch is not handled by this ingredient (see `CommandTreeIngredient`).

The second ingredient is *AutocompleteIngredient* which relies on the third-party argcomplete module to add support for automatic command line completion to supported shells (bash).

class guacamole.ingredients.argparse.**AutocompleteIngredient**
Ingredient for adding shell auto-completion.

> **Warning:** This component is not widely tested due to difficulty of providing actual integration. It might be totally broken.

Note: To effectively get tab completion you need to have the `argcomplete` package installed. In addition, a per-command initialization command has to be created and sourced by the shell. Look at argcomplete documentation for details.

---

parse(*context*)
Optionally trigger argument completion in the invoking shell.

This method is called to see if bash argument completion is requested and to honor the request, if needed. This causes the process to exit (early) without giving other ingredients a chance to initialize or shut down.

Due to the way argcomple works, no other ingredient can print() anything to stdout prior to this point.

class guacamole.ingredients.argparse.**ParserIngredient**
Ingredient for using argparse to parse command line arguments.

This ingredient uses the following Ingredient methods:

- `build_early_parser()`
- `preparse()`
- `build_parser()`
- `parse()`

The main parser is constructed in, unsurprisingly, the *build_parser()* method and stored in the context as `parser`. Other ingredients can be added *after* the `ParserIngredient` and can extend the available arguments (on the root parser) by using standard argparse APIs such as `parser.add_argument()` or `parser.add_argument_group()`. This parser is used to handle all of command line in the *parse()* method.

While most users won't concern themselves with this design decision, there is also a second parser, called the *early parser*, that is used to *pre-parse* the command line arguments. This can be used as a way to optimize subsequent actions as, perhaps, knowing which commands are going to be invoked there will be no need to instantiate and prepare *all* of the commands in the command tree.

Currently this feature is not used. To take advantage of this knowledge you can look at the `context.early_args` object which contains the result of parsing the command line with the *early parser*. The early parser is a simple parser consisting of `--help`, `--version` (if applicable) and *rest*. The *rest* argument can be used as a hint as to what is coming next (e.g. if it matches a name of a command we know to exist)

After parsing is done the results of parsing the command line are stored in the `context.args` attribute. This is commonly accessed by individual commands from their `invoke()` methods.

build_early_parser(*context*)
Create the early argument parser.

This method creates the early argparse argument parser. The early parser doesn't know about any of the sub-commands so it can be used much earlier during the start-up process (before commands are loaded and initialized).

build_parser(*context*)
Create the final argument parser.

This method creates the non-early (full) argparse argument parser. Unlike the early counterpart it is expected to have knowledge of the full command tree.

This method relies on `context.cmd_tree` and produces `context.parser`. Other ingredients can interact with the parser up until *parse()* is called.

parse(*context*)
Parse command line arguments.

This method relies on `context.argv` and `context.early_parser` and produces `context.args`. Note that `.argv` is modified by *preparse()* so it actually has _less_ things in it.

The `context.args` object is the return value from argparse. It is the dict/object like namespace object.

preparse(*context*)
Parse a portion of command line arguments with the early parser.

This method relies on `context.argv` and `context.early_parser` and produces `context.early_args`.

The `context.early_args` object is the return value from argparse. It is the dict/object like namespace object.

8.7 guacamole.ingredients.crash

Ingredient for reacting to application crashes.

class guacamole.ingredients.crash.**VerboseCrashHandler**
Ingredient for reacting to crashes with a traceback.

You can add this ingredient into your recipe to react to application crashes. It will simply print the exception, as stored in `context.exc_type`, `context.exc_value` and `context.traceback` and raise SystemExit(1).

dispatch_failed(*context*)
Print the unhandled exception and exit the application.

8.8 guacamole.ingredients.ansi

Ingredients for using ANSI command sequences.

class guacamole.ingredients.ansi.**ANSI**
Numerous ANSI constants.

See also:

Original specification in the Standard ECMA 48, page 61 (75th page of the PDF).

Wikipedia article about ANSI escape code.

cmd_erase_display
Command for erasing the whole display

cmd_erase_line
Command for erasing the current line

cmd_sgr_reset_all:
Command for resetting all SGR attributes

sgr_reset_all:
SGR code for resetting all attributes

sgr_bold:
Causes text to be rendered with bold face font or, alternatively, to be rendered with bright color variant. This code is widely supported on Linux. It is not supported on Windows.

sgr_bright:
Alternate spelling of :attr:`sgr_bold`.

sgr_faint:
SGR code that activates faint color subset

srg_dim:
Alternate spelling of ``sgr_faint``

srg_italic:
SGR code that activates italic font face

sgr_underline:
SGR code that activates underline mode

sgr_blink_slow:
SGR code that activates slow blinking of characters

sgr_blink_fast:
SGR code that activates fast blinking of characters

sgr_reverse:
SGR code that activates reverse-video mode

sgr_double_underline:
SGR code that activates double-underline mode

static cmd_sgr(*sgr_list*)
Get a SGR (Set Graphics Rendition) code.

static sgr_bg_indexed(*i*)
Get SGR (Set Graphics Rendition) background indexed color.

static sgr_bg_rgb(*r*, *g*, *b*)
Get SGR (Set Graphics Rendition) background RGB color.

static sgr_fg_indexed(*i*)
Get SGR (Set Graphics Rendition) foreground indexed color.

static sgr_fg_rgb(*r*, *g*, *b*)
Get SGR (Set Graphics Rendition) foreground RGB color.

class guacamole.ingredients.ansi.**ANSIFormatter**(*enabled=None*)
Formatter for ANSI Set Graphics Rendition codes.

An instance of this class is inserted into the context object as *ansi*. Using the fact that *ANSIFormatter* is callable one can easily add ANSI control sequences for foreground and background color as well as text attributes.

aprint(**values*, ***kwargs*)
ANSI formatting-aware print().

This method is a version of print() (function) that understands additional ansi control parameters.

Parameters

- **value** – The values to print, same as with `print()`
- **sep** – Separator between values, same as with `print()`
- **end** – Terminator of the line, same as with `print()`
- **file** – File to print to, same as with `print()`
- **flush** – Flag that controls stream flush behavior, same as with `print()`
- **fg** – Foreground color, same as with `__call__()`.
- **bg** – Background color, same as with `__call__()`.
- **style** – Text style, same as with `__call__()`.

- **reset** – Flag that controls if ANSI attributes are reset at the end, same as with `__call__()`.
- **sgr** – Additonal (custom) Set Graphics Rendition directives, same as with `__call__()`.

Note: This implementation only works on Python 3

cmd(*cmd*, **args*)
Get an ANSI control sequence, if the formatter is enabled.

is_enabled
Flag indicating if text style is enabled.

This property is useful to let applications customize their behavior if they know color support is desired and enabled.

class guacamole.ingredients.ansi.**ANSIIngredient**(*enable=None*)
Ingredient for colorizing output.

added(*context*)
Ingredient method called before anything else.

guacamole.ingredients.ansi.**ansi_cmd**(*cmd*, **args*)
Get ANSI command code by name.

guacamole.ingredients.ansi.**ansi_sgr**(*text*, *fg=None*, *bg=None*, *style=None*, *reset=True*, ***sgr*)
Apply desired SGR commands to given text.

Parameters

- **text** – Text or anything convertible to text
- **fg** – (optional) Foreground color. Choose one of `black`, `red`, `green`, `yellow`, `blue`, `magenta` `cyan` or `white`. Note that the `bright` *SGR* impacts effective color in most implementations.

guacamole.ingredients.ansi.**get_intensity**(*r*, *g*, *b*)
Get the gray level intensity of the given rgb triplet.

guacamole.ingredients.ansi.**get_visible_color**(*color*)
Get the visible counter-color.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

g

A

B

C

D

E

G

H

I

L

M

P

R

S

V