rudiments Documentation

Release 0.4.1

Jürgen Hermann

2020-03-27

CONTENTS

Rudiments is a Python library that offers 'miscellaneous' functionality which is unspecific in nature and shared among many projects. It also collects small extensions to other support packages that don't warrant their own project, in the `rudiments.reamed` package.

CHAPTER

ONE

IMPORTANT LINKS

- GitHub Project
- Issue Tracker
- PyPI
- Latest Documentation
- Mailing List

CHAPTER TWO

INSTALLING

Rudiments can be installed from PyPI via `pip install rudiments` as usual, see releases on GitHub for an overview of available versions – the project uses semantic versioning and follows PEP 440 conventions. To get a bleeding-edge version from source, use these commands:

```
repo="jhermann/rudiments"
pip install -r "https://raw.githubusercontent.com/$repo/master/requirements.txt"
pip install -UI -e "git+https://github.com/$repo.git#egg=${repo#*/}"
```

See the following section on how to create a full development environment.

CHAPTER THREE

CONTRIBUTING

To create a working directory for this project, call these commands:

```
git clone "https://github.com/jhermann/rudiments.git"
cd "rudiments"
. .env --yes --develop
invoke build --docs test check
```

Contributing to this project is easy, and reporting an issue or adding to the documentation also improves things for every user. You don’t need to be a developer to contribute. See *Contribution Guidelines* for more.

CHAPTER

FOUR

DOCUMENTATION CONTENTS

4.1 Using rudiments

4.1.1 Web Access Helpers

The *rudiments.www* module helps with handling web resources.

The context manager *rudiments.www.url_as_file()* can be used to make the content of an URL available as a local file, so it can be fed to things that only work with local filesystem paths. House-keeping is automatic, so the file is removed on leaving the context unless you removed or moved it yourself before that.

4.1.2 Security Helpers

4.1.2.1 Credentials Lookup

When using HTTP APIs or other secured web resources, you are confronted with the question how to enable your users to store their credentials in a *secure but still convenient* fashion. The *rudiments.security.Credentials* class tries to give an answer, by providing some common methods for credential lookup that occupy different spots in the secure vs. convenient spectrum. See *Configuration of Authentication Credentials* for details regarding usage from the viewpoint of an end-user and some information about the availabe credential providers.

To use the class, create a *rudiments.security.Credentials* object, passing in the *target*. Then to retrieve matching credentials, call the *rudiments.security.Credentials.auth_pair()* method.

```
access = Credentials('http://jane@doe.example.com')
username, password = access.auth_pair()
```

Note that this allows to only prompt the user for a password when it's actually needed, but still create the credentials object early on, during some setup phase.

4.1.3 Humanized Input and Output

For accepting input from prompts and configuration files, and presenting values in a form easily parsed by humans, the *rudiments.humanize* module offers conversion functions for common data types.

For handling byte sizes in IEC binary units, use *rudiments.humanize.bytes2iec()* and *rudiments.humanize.iec2bytes()*. Examples:

```
>>> bytes2iec(1536), bytes2iec(10**9)
(u'   1.5 KiB', u' 953.7 MiB')
>>> bytes2iec(1536, compact=True)
u'1.5KiB'
>>> iec2bytes(1), iec2bytes('64k'), iec2bytes('1.234TiB')
(1, 65536, 1356797348675)
```

By default, the formatted values are suited for tabulated output (they're all the same length); when passing `compact=True`, you'll get a result that better fits into log messages.

To present lists of numbers in a compact form, collapsing consecutive ranges, *`rudiments.humanize.merge_adjacent()`* can be used.

```
>>> ', '.join(humanize.merge_adjacent(('9', 5, 10, 7) + tuple(range(5))))
u'0..5, 7, 9..10'
```

4.1.4 Python Runtime Support

Use the *`rudiments.pysupport`* module to access some helpers which hide internals of the Python interpreter runtime and provide an easier to use interface.

The functions *`rudiments.pysupport.import_name()`* and *`rudiments.pysupport.load_module()`* can be used for dynamic imports and adding a simple plug-in system to your application.

To help with keeping code portable between Python 2.7 and 3.x, the `rudiments._compat` module offers unified names and semantics for common features that differ between current and legacy Python versions. It is based on the module with the same name found in Jinja2.

4.1.5 Operating System Related Extensions

In *`rudiments.system`*, you find low-level extensions to stdlib modules like `os` and `sys`.

Constants in this module that start with `EX_` are standard exit codes to be used with `sys.exit()`, as defined in the C header file `sysexits.h`.

4.1.6 Extensions to 3rd Party Libraries

The sub-package *`rudiments.reamed`* contains modules that extend the API of some outside library.

Note that you need to add the underlying package to your dependencies in addition to rudiments, in case you use one of the modules in that sub-package. `rudiments` itself does not publish any dependencies on them.

Where the extended package has a condensed public API (i.e. names are usually only imported from the package name), these modules can serve as a drop-in replacement, so you just have to change the import statement a little.

4.1.7 Extensions to Click

You can use the *rudiments.reamed.click* module as a drop-in replacement for Click, like this:

```
from rudiments.reamed import click
```

There are additional helper functions: *rudiments.reamed.click.pretty_path()* wraps *rudiments.reamed.click.format_filename()* to make a file system path presentable to humans, especially for logging purposes. The *rudiments.reamed.click.serror()* function prints an already styled, very visible error message, while using any arguments to format the message.

The *rudiments.reamed.click.LoggedFailure* exception can be used when you want to abort a command with a clearly visible error – the message is styled identically to what `serror()` produces, white bold text on a red background.

rudiments.reamed.click.AliasedGroup allows you to define alias names for commands you defined via the usual annotatons. Here is an example that maps the `ls` alias to the official `list` command name:

```
from rudiments.reamed import click

class SmurfAliases(click.AliasedGroup):
    """Alias mapping for 'smurf' commands."""
    MAP = dict(
        ls='list',
    )


@cli.group(cls=SmurfAliases)
def smurf():
    """Management of smurfs."""


@smurf.command(name='list')
def smurf_list():
    """A command that lists smurfs."""
    # ...
```

Finally, the biggest addition is a default configuration parsing machinery in the *rudiments.reamed.click.Configuration* class. It should be instantiated in your root command, passing in the (optional) name of a specific configuration file, or a path of such files.

```
@click.group()
@click.option('-c', '--config', "config_paths", metavar='FILE',
              multiple=True, type=click.Path(), help='Load given configuration␣
↪file(s).')
@click.pass_context
def cli(ctx, config_paths=None):
    """Some command line tool."""
    config.Configuration.from_context(ctx, config_paths)
```

The prepared configuration object is then available to any sub-command via the context, as `ctx.obj.cfg`. For more details, see the *rudiments.reamed.click.Configuration* documentation.

4.2 End-User Documentation

This chapter contains instructions targeted at users of projects that are using this library, so that you can link to thse from your onw documentation.

4.2.1 Configuration of Authentication Credentials

4.2.1.1 Credentials Lookup Details

When using HTTP APIs or other secured web resources, you usually want to store your credentials in a *secure but still convenient* fashion. Given a *target* that requires authentication in the form of a username and password or API token, the application will try several methods to find matching credentials in 'common' places.

For URLs (`http`, `https`, `ftp`, or `ftps`), the following steps will be taken:

- The URL's `user@pwd` part is checked first and used if present.
- Next, the system's keyring is queried for an entry under the URL's host name.
- Similarly, `~/.netrc` is scanned for matching entries next.
- If nothing can be found, you will be prompted on the console.

As a general fallback, any given target that is not an URL will ask for a username / password pair.

The keyring and netrc file are queried for an entry matching the hostname and account name, with the latter being taken from the URL if present, else the user's login name is used. This allows you to easily assume different roles on a target system, e.g. to access a normal and a privileged account. So for an admin account, use something like `https://admin@service.example.com/` and a matching password entry for `admin` on `service.example.com`.

In netrc files, the `machine` entries must be unique, so the name `user@host` is queried before the plain host name. This way you can provide credentials for several accounts on the same target in one file.

4.2.1.2 Installation Procedures

For using netrc files and prompting, nothing extra has to be installed, because Python has everything needed on board. By using keyring credentials, you gain more security (stored passwords are encrypted and only available after you logged in to your account), at the possible price of installing additional software. Consult the manual of your application whether any of the following installation steps are actually necessary and suitable — at least the `keyring` Python package will normally be included when you install an application.

On *Windows* and *Mac OS X*, you don't need to install extra system software, but on a *Linux* system the OS package necessary for installing the `dbus-python` Python package has to be made available. On Debian-type systems, that means calling this command:

```
sudo apt-get install libdbus-glib-1-dev python-dev libffi-dev build-essential
```

For the Python packages, use `pip` as follows:

```
pip install secretstorage dbus-python keyring
```

For *Windows* and *Mac OS X*, only `keyring` is needed. To test that you installed all supporting libraries in a *Linux* setup, try this:

```
$ python -c "import keyring; print keyring.get_keyring()"
<keyring.backends.SecretService.Keyring object at 0x7f091526bcd0>
```

If it doesn't work or the essential components are not installed, in the output you'll get `keyring.backends.fail.Keyring` instead. A successful installation on other operating systems will show some different back-end that is not the 'fail' one.

On a *Gnome* desktop (e.g. *Ubuntu 14.04* and up), the end-user application to manage passwords is `seahorse` a/k/a "Passwords and Keys". It can be used to check that your passwords are stored correctly, and to change and delete them.

4.3 Changelog

- : Windows path handling fixed
- : Newer Click versions are now supported
- : Keyring support is disabled for now (was never very stable anyway)
- : Supported Python versions were limited to 3.5 … 3.8
- : replaced *bunch* with *munch* (which is maintained)
- : click config: add *optional* param to *section()*
- : Credentials: fall back to console prompt for anything not an URL
- : new: *rudiments.security* module
- : new: this changelog
- : bug: need to sort any config.d listing
- : chg: added '/etc/{appname}.d/' to default config paths
- : new: 'system' module with standard process exit codes
- : new: 'pysupport' module with import helpers
- : new: 'humanize' module for I/O of common values in forms understood by humans
- : new: Sphinx documentation at https://rudiments.readthedocs.org/
- : *reamed.click* extensions for Click
- : *www.url_as_file* context manager
- : TODO add missing items

4.4 Complete API Reference

The following is a complete API reference generated from source.

4.4.1 rudiments package

Rudiments – Fundamental elements for any Python project.

This package offers configuration handling and other basic helpers for Python projects.

4.4.1.1 Subpackages

rudiments.reamed package

Extensions to third-party libraries.

Note that you need to add the underlying package to your dependencies in addition to `rudiments`, in case you use one of the modules in here. `rudiments` itself does not publish any dependencies on them.

Where the extended package has a condensed public API (i.e. names are usually only imported from the package name), these modules can serve as a drop-in replacement, so you just have to change the import statement a little.

Submodules

rudiments.reamed.click module

'Double Click' – Extensions to Click.

exception `rudiments.reamed.click.`**`Abort`**

Bases: `RuntimeError`

An internal signalling exception that signals Click to abort.

class `rudiments.reamed.click.`**`Argument`**(*param_decls*, *required=None*, ***attrs*)

Bases: `click.core.Parameter`

Arguments are positional parameters to a command. They generally provide fewer features than options but can have infinite `nargs` and are required by default.

All parameters are passed onwards to the parameter constructor.

`add_to_parser`(*parser*, *ctx*)

`get_error_hint`(*ctx*)

Get a stringified version of the param for use in error messages to indicate which param caused the error.

`get_usage_pieces`(*ctx*)

property human_readable_name

Returns the human readable name of this parameter. This is the same as the name for options, but the metavar for arguments.

`make_metavar`()

param_type_name = 'argument'

exception `rudiments.reamed.click.`**BadArgumentUsage**(*message*, *ctx=None*)

Bases: `click.exceptions.UsageError`

Raised if an argument is generally supplied but the use of the argument was incorrect. This is for instance raised if the number of values for an argument is not correct.

New in version 6.0.

exception `rudiments.reamed.click.`**BadOptionUsage**(*option_name*, *message*, *ctx=None*)

Bases: `click.exceptions.UsageError`

Raised if an option is generally supplied but the use of the option was incorrect. This is for instance raised if the number of arguments for an option is not correct.

New in version 4.0.

Parameters **option_name** – the name of the option being used incorrectly.

exception `rudiments.reamed.click.`**BadParameter**(*message*, *ctx=None*, *param=None*, *param_hint=None*)

Bases: `click.exceptions.UsageError`

An exception that formats out a standardized error message for a bad parameter. This is useful when thrown from a callback or type as Click will attach contextual information to it (for instance, which parameter it is).

New in version 2.0.

Parameters

- **param** – the parameter object that caused this error. This can be left out, and Click will attach this info itself if possible.
- **param_hint** – a string that shows up as parameter name. This can be used as alternative to *param* in cases where custom validation should happen. If it is a string it's used as such, if it's a list then each item is quoted and separated.

format_message()

class `rudiments.reamed.click.`**BaseCommand**(*name*, *context_settings=None*)

Bases: `object`

The base command implements the minimal API contract of commands. Most code will never use this as it does not implement a lot of useful functionality but it can act as the direct subclass of alternative parsing methods that do not depend on the Click parser.

For instance, this can be used to bridge Click and other systems like argparse or docopt.

Because base commands do not implement a lot of the API that other parts of Click take for granted, they are not supported for all operations. For instance, they cannot be used with the decorators usually and they have no built-in callback system.

Changed in version 2.0: Added the *context_settings* parameter.

Parameters

- **name** – the name of the command to use unless a group overrides it.
- **context_settings** – an optional dictionary with defaults that are passed to the context object.

allow_extra_args = False

the default for the `Context.allow_extra_args` flag.

allow_interspersed_args = True
the default for the `Context.allow_interspersed_args` flag.

context_settings = None
an optional dictionary with defaults passed to the context.

get_help(*ctx*)

get_usage(*ctx*)

ignore_unknown_options = False
the default for the `Context.ignore_unknown_options` flag.

invoke(*ctx*)
Given a context, this invokes the command. The default implementation is raising a not implemented error.

main(*args=None*, *prog_name=None*, *complete_var=None*, *standalone_mode=True*, ***extra*)
This is the way to invoke a script with all the bells and whistles as a command line application. This will always terminate the application after a call. If this is not wanted, `SystemExit` needs to be caught.

This method is also available by directly calling the instance of a `Command`.

New in version 3.0: Added the *standalone_mode* flag to control the standalone mode.

Parameters

- **args** – the arguments that should be used for parsing. If not provided, `sys.argv[1:]` is used.
- **prog_name** – the program name that should be used. By default the program name is constructed by taking the file name from `sys.argv[0]`.
- **complete_var** – the environment variable that controls the bash completion support. The default is `"_<prog_name>_COMPLETE"` with prog_name in uppercase.
- **standalone_mode** – the default behavior is to invoke the script in standalone mode. Click will then handle exceptions and convert them into error messages and the function will never return but shut down the interpreter. If this is set to *False* they will be propagated to the caller and the return value of this function is the return value of `invoke()`.
- **extra** – extra keyword arguments are forwarded to the context constructor. See `Context` for more information.

make_context(*info_name*, *args*, *parent=None*, ***extra*)
This function when given an info name and arguments will kick off the parsing and create a new `Context`. It does not invoke the actual command callback though.

Parameters

- **info_name** – the info name for this invokation. Generally this is the most descriptive name for the script or command. For the toplevel script it's usually the name of the script, for commands below it it's the name of the script.
- **args** – the arguments to parse as list of strings.
- **parent** – the parent context if available.
- **extra** – extra keyword arguments forwarded to the context constructor.

name = None
the name the command thinks it has. Upon registering a command on a `Group` the group will default the command name with this information. You should instead use the `Context`'s `info_name` attribute.

parse_args (*ctx*, *args*)
Given a context and a list of arguments this creates the parser and parses the arguments, then modifies the context as necessary. This is automatically invoked by `make_context()`.

class `rudiments.reamed.click.`**Choice** (*choices*, *case_sensitive=True*)
Bases: `click.types.ParamType`

The choice type allows a value to be checked against a fixed set of supported values. All of these values have to be strings.

You should only pass a list or tuple of choices. Other iterables (like generators) may lead to surprising results.

The resulting value will always be one of the originally passed choices regardless of `case_sensitive` or any `ctx.token_normalize_func` being specified.

See choice-opts for an example.

Parameters **case_sensitive** – Set to false to make choices case insensitive. Defaults to true.

convert (*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

get_metavar (*param*)
Returns the metavar default for this param if it provides one.

get_missing_message (*param*)
Optionally might return extra information about a missing parameter.

New in version 2.0.

name = 'choice'

exception `rudiments.reamed.click.`**ClickException** (*message*)
Bases: `Exception`

An exception that Click can handle and show to the user.

exit_code = 1
The exit code for this exception

format_message ()

show (*file=None*)

class `rudiments.reamed.click.`**Command** (*name*, *context_settings=None*, *callback=None*, *params=None*, *help=None*, *epilog=None*, *short_help=None*, *options_metavar='[OPTIONS]'*, *add_help_option=True*, *no_args_is_help=False*, *hidden=False*, *deprecated=False*)
Bases: `click.core.BaseCommand`

Commands are the basic building block of command line interfaces in Click. A basic command handles command line parsing and might dispatch more parsing to commands nested below it.

Changed in version 2.0: Added the *context_settings* parameter.

Changed in version 7.1: Added the *no_args_is_help* parameter.

Parameters

- **name** – the name of the command to use unless a group overrides it.
- **context_settings** – an optional dictionary with defaults that are passed to the context object.
- **callback** – the callback to invoke. This is optional.

- **params** – the parameters to register with this command. This can be either *Option* or *Argument* objects.
- **help** – the help string to use for this command.
- **epilog** – like the help string but it's printed at the end of the help page after everything else.
- **short_help** – the short help to use for this command. This is shown on the command listing of the parent command.
- **add_help_option** – by default each command registers a `--help` option. This can be disabled by this parameter.
- **no_args_is_help** – this controls what happens if no arguments are provided. This option is disabled by default. If enabled this will add `--help` as argument if no arguments are passed
- **hidden** – hide this command from help outputs.
- **deprecated** – issues a message indicating that the command is deprecated.

callback = None
the callback to execute when the command fires. This might be *None* in which case nothing happens.

collect_usage_pieces(*ctx*)
Returns all the pieces that go into the usage line and returns it as a list of strings.

format_epilog(*ctx, formatter*)
Writes the epilog into the formatter if it exists.

format_help(*ctx, formatter*)
Writes the help into the formatter if it exists.

This is a low-level method called by *get_help()*.

This calls the following methods:

- *format_usage()*
- *format_help_text()*
- *format_options()*
- *format_epilog()*

format_help_text(*ctx, formatter*)
Writes the help text to the formatter if it exists.

format_options(*ctx, formatter*)
Writes all the options into the formatter if they exist.

format_usage(*ctx, formatter*)
Writes the usage line into the formatter.

This is a low-level method called by *get_usage()*.

get_help(*ctx*)
Formats the help into a string and returns it.

Calls *format_help()* internally.

get_help_option(*ctx*)
Returns the help option object.

get_help_option_names(*ctx*)
Returns the names for the help option.

get_params(*ctx*)

get_short_help_str(*limit=45*)
Gets short help for the command or makes it by shortening the long help string.

get_usage(*ctx*)
Formats the usage line into a string and returns it.

Calls *format_usage()* internally.

invoke(*ctx*)
Given a context, this invokes the attached callback (if it exists) in the right way.

make_parser(*ctx*)
Creates the underlying option parser for this command.

params = None
the list of parameters for this command in the order they should show up in the help page and execute. Eager parameters will automatically be handled before non eager ones.

parse_args(*ctx*, *args*)
Given a context and a list of arguments this creates the parser and parses the arguments, then modifies the context as necessary. This is automatically invoked by `make_context()`.

class `rudiments.reamed.click.`**CommandCollection**(*name=None*, *sources=None*, ***attrs*)
Bases: `click.core.MultiCommand`

A command collection is a multi command that merges multiple multi commands together into one. This is a straightforward implementation that accepts a list of different multi commands as sources and provides all the commands for each of them.

add_source(*multi_cmd*)
Adds a new multi command to the chain dispatcher.

get_command(*ctx*, *cmd_name*)
Given a context and a command name, this returns a *Command* object if it exists or returns *None*.

list_commands(*ctx*)
Returns a list of subcommand names in the order they should appear.

sources = None
The list of registered multi commands.

class `rudiments.reamed.click.`**Context**(*command*, *parent=None*, *info_name=None*, *obj=None*, *auto_envvar_prefix=None*, *default_map=None*, *terminal_width=None*, *max_content_width=None*, *resilient_parsing=False*, *allow_extra_args=None*, *allow_interspersed_args=None*, *ignore_unknown_options=None*, *help_option_names=None*, *token_normalize_func=None*, *color=None*, *show_default=None*)
Bases: `object`

The context is a special internal object that holds state relevant for the script execution at every single level. It's normally invisible to commands unless they opt-in to getting access to it.

The context is useful as it can pass internal objects around and can control special execution features such as reading data from environment variables.

A context can be used as context manager in which case it will call `close()` on teardown.

New in version 2.0: Added the *resilient_parsing*, *help_option_names*, *token_normalize_func* parameters.

New in version 3.0: Added the *allow_extra_args* and *allow_interspersed_args* parameters.

New in version 4.0: Added the *color*, *ignore_unknown_options*, and *max_content_width* parameters.

New in version 7.1: Added the *show_default* parameter.

Parameters

- **`command`** – the command class for this context.
- **`parent`** – the parent context.
- **`info_name`** – the info name for this invocation. Generally this is the most descriptive name for the script or command. For the toplevel script it is usually the name of the script, for commands below it it's the name of the script.
- **`obj`** – an arbitrary object of user data.
- **`auto_envvar_prefix`** – the prefix to use for automatic environment variables. If this is *None* then reading from environment variables is disabled. This does not affect manually set environment variables which are always read.
- **`default_map`** – a dictionary (like object) with default values for parameters.
- **`terminal_width`** – the width of the terminal. The default is inherit from parent context. If no context defines the terminal width then auto detection will be applied.
- **`max_content_width`** – the maximum width for content rendered by Click (this currently only affects help pages). This defaults to 80 characters if not overridden. In other words: even if the terminal is larger than that, Click will not format things wider than 80 characters by default. In addition to that, formatters might add some safety mapping on the right.
- **`resilient_parsing`** – if this flag is enabled then Click will parse without any interactivity or callback invocation. Default values will also be ignored. This is useful for implementing things such as completion support.
- **`allow_extra_args`** – if this is set to *True* then extra arguments at the end will not raise an error and will be kept on the context. The default is to inherit from the command.
- **`allow_interspersed_args`** – if this is set to *False* then options and arguments cannot be mixed. The default is to inherit from the command.
- **`ignore_unknown_options`** – instructs click to ignore options it does not know and keeps them for later processing.
- **`help_option_names`** – optionally a list of strings that define how the default help parameter is named. The default is `['--help']`.
- **`token_normalize_func`** – an optional function that is used to normalize tokens (options, choices, etc.). This for instance can be used to implement case insensitive behavior.
- **`color`** – controls if the terminal supports ANSI colors or not. The default is autodetection. This is only needed if ANSI codes are used in texts that Click prints which is by default not the case. This for instance would affect help output.
- **`show_default`** – if True, shows defaults for all options. Even if an option is later created with show_default=False, this command-level setting overrides it.

`abort``()`
Aborts the script.

allow_extra_args = None
Indicates if the context allows extra args or if it should fail on parsing.

New in version 3.0.

allow_interspersed_args = None
Indicates if the context allows mixing of arguments and options or not.

New in version 3.0.

args = None
the leftover arguments.

call_on_close(*f*)
This decorator remembers a function as callback that should be executed when the context tears down. This is most useful to bind resource handling to the script execution. For instance, file objects opened by the *File* type will register their close callbacks here.

Parameters **f** – the function to execute on teardown.

close()
Invokes all close callbacks.

color = None
Controls if styling output is wanted or not.

command = None
the *Command* for this context.

property command_path
The computed command path. This is used for the `usage` information on the help page. It's automatically created by combining the info names of the chain of contexts to the root.

ensure_object(*object_type*)
Like *find_object()* but sets the innermost object to a new instance of *object_type* if it does not exist.

exit(*code=0*)
Exits the application with a given exit code.

fail(*message*)
Aborts the execution of the program with a specific error message.

Parameters **message** – the error message to fail with.

find_object(*object_type*)
Finds the closest object of a given type.

find_root()
Finds the outermost context.

forward(***kwargs*)
Similar to *invoke()* but fills in default keyword arguments from the current context if the other command expects it. This cannot invoke callbacks directly, only other commands.

get_help()
Helper method to get formatted help page for the current context and command.

get_usage()
Helper method to get formatted usage string for the current context and command.

help_option_names = None
The names for the help options.

ignore_unknown_options = None

Instructs click to ignore options that a command does not understand and will store it on the context for later processing. This is primarily useful for situations where you want to call into external programs. Generally this pattern is strongly discouraged because it's not possibly to losslessly forward all arguments.

New in version 4.0.

info_name = None

the descriptive information name

invoke(***kwargs*)

Invokes a command callback in exactly the way it expects. There are two ways to invoke this method:

1. the first argument can be a callback and all other arguments and keyword arguments are forwarded directly to the function.
2. the first argument is a click command object. In that case all arguments are forwarded as well but proper click parameters (options and click arguments) must be keyword arguments and Click will fill in defaults.

Note that before Click 3.2 keyword arguments were not properly filled in against the intention of this code and no context was created. For more information about this change and why it was done in a bugfix release see upgrade-to-3.2.

invoked_subcommand = None

This flag indicates if a subcommand is going to be executed. A group callback can use this information to figure out if it's being executed directly or because the execution flow passes onwards to a subcommand. By default it's None, but it can be the name of the subcommand to execute.

If chaining is enabled this will be set to `'*'` in case any commands are executed. It is however not possible to figure out which ones. If you require this knowledge you should use a `resultcallback()`.

lookup_default(*name*)

Looks up the default for a parameter name. This by default looks into the `default_map` if available.

make_formatter()

Creates the formatter for the help and usage output.

max_content_width = None

The maximum width of formatted content (None implies a sensible default which is 80 for most things).

property meta

This is a dictionary which is shared with all the contexts that are nested. It exists so that click utilities can store some state here if they need to. It is however the responsibility of that code to manage this dictionary well.

The keys are supposed to be unique dotted strings. For instance module paths are a good choice for it. What is stored in there is irrelevant for the operation of click. However what is important is that code that places data here adheres to the general semantics of the system.

Example usage:

```
LANG_KEY = f'{__name__}.lang'

def set_language(value):
    ctx = get_current_context()
    ctx.meta[LANG_KEY] = value

def get_language():
    return get_current_context().meta.get(LANG_KEY, 'en_US')
```

New in version 5.0.

obj = None
the user object stored.

params = None
the parsed parameters except if the value is hidden in which case it's not remembered.

parent = None
the parent context or *None* if none exists.

protected_args = None
protected arguments. These are arguments that are prepended to *args* when certain parsing scenarios are encountered but must be never propagated to another arguments. This is used to implement nested parsing.

resilient_parsing = None
Indicates if resilient parsing is enabled. In that case Click will do its best to not cause any failures and default values will be ignored. Useful for completion.

scope(*cleanup=True*)
This helper method can be used with the context object to promote it to the current thread local (see *get_current_context()*). The default behavior of this is to invoke the cleanup functions which can be disabled by setting *cleanup* to *False*. The cleanup functions are typically used for things such as closing file handles.

If the cleanup is intended the context object can also be directly used as a context manager.

Example usage:

```
with ctx.scope():
    assert get_current_context() is ctx
```

This is equivalent:

```
with ctx:
    assert get_current_context() is ctx
```

New in version 5.0.

Parameters **cleanup** – controls if the cleanup functions should be run or not. The default is to run these functions. In some situations the context only wants to be temporarily pushed in which case this can be disabled. Nested pushes automatically defer the cleanup.

terminal_width = None
The width of the terminal (None is autodetection).

token_normalize_func = None
An optional normalization function for tokens. This is options, choices, commands etc.

class rudiments.reamed.click.**DateTime**(*formats=None*)
Bases: `click.types.ParamType`

The DateTime type converts date strings into *datetime* objects.

The format strings which are checked are configurable, but default to some common (non-timezone aware) ISO 8601 formats.

When specifying *DateTime* formats, you should only pass a list or a tuple. Other iterables, like generators, may lead to surprising results.

The format strings are processed using `datetime.strptime`, and this consequently defines the format strings which are allowed.

Parsing is tried using each format, in order, and the first format which parses successfully is used.

Parameters **`formats`** – A list or tuple of date format strings, in the order in which they should be tried. Defaults to `'%Y-%m-%d'`, `'%Y-%m-%dT%H:%M:%S'`, `'%Y-%m-%d %H:%M:%S'`.

`convert`(*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

`get_metavar`(*param*)
Returns the metavar default for this param if it provides one.

`name = 'datetime'`

class `rudiments.reamed.click.`**`File`**(*mode='r'*, *encoding=None*, *errors='strict'*, *lazy=None*, *atomic=False*)
Bases: `click.types.ParamType`

Declares a parameter to be a file for reading or writing. The file is automatically closed once the context tears down (after the command finished working).

Files can be opened for reading or writing. The special value - indicates stdin or stdout depending on the mode.

By default, the file is opened for reading text data, but it can also be opened in binary mode or for writing. The encoding parameter can be used to force a specific encoding.

The *lazy* flag controls if the file should be opened immediately or upon first IO. The default is to be non-lazy for standard input and output streams as well as files opened for reading, *lazy* otherwise. When opening a file lazily for reading, it is still opened temporarily for validation, but will not be held open until first IO. lazy is mainly useful when opening for writing to avoid creating the file until it is needed.

Starting with Click 2.0, files can also be opened atomically in which case all writes go into a separate file in the same folder and upon completion the file will be moved over to the original location. This is useful if a file regularly read by other users is modified.

See file-args for more information.

`convert`(*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

`envvar_list_splitter = ':'`

`name = 'filename'`

`resolve_lazy_flag`(*value*)

exception `rudiments.reamed.click.`**`FileError`**(*filename*, *hint=None*)
Bases: `click.exceptions.ClickException`

Raised if a file cannot be opened.

`format_message`()

class `rudiments.reamed.click.`**`FloatRange`**(*min=None*, *max=None*, *clamp=False*)
Bases: `click.types.FloatParamType`

A parameter that works similar to `click.FLOAT` but restricts the value to fit into a range. The default behavior is to fail if the value falls outside the range, but it can also be silently clamped between the two edges.

See ranges for an example.

`convert`(*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

`name = 'float range'`

class `rudiments.reamed.click.`**`Group`**(*name=None*, *commands=None*, ***attrs*)
Bases: `click.core.MultiCommand`

A group allows a command to have subcommands attached. This is the most common way to implement nesting in Click.

Parameters **commands** – a dictionary of commands.

add_command(*cmd*, *name=None*)
Registers another `Command` with this group. If the name is not provided, the name of the command is used.

command(**args*, ***kwargs*)
A shortcut decorator for declaring and attaching a command to the group. This takes the same arguments as `command()` but immediately registers the created command with this instance by calling into `add_command()`.

commands = None
the registered subcommands by their exported names.

get_command(*ctx*, *cmd_name*)
Given a context and a command name, this returns a `Command` object if it exists or returns *None*.

group(**args*, ***kwargs*)
A shortcut decorator for declaring and attaching a group to the group. This takes the same arguments as `group()` but immediately registers the created command with this instance by calling into `add_command()`.

list_commands(*ctx*)
Returns a list of subcommand names in the order they should appear.

class rudiments.reamed.click.**HelpFormatter**(*indent_increment=2*, *width=None*, *max_width=None*)
Bases: `object`

This class helps with formatting text-based help pages. It's usually just needed for very special internal cases, but it's also exposed so that developers can write their own fancy outputs.

At present, it always writes into memory.

Parameters

- **indent_increment** – the additional increment for each level.
- **width** – the width for the text. This defaults to the terminal width clamped to a maximum of 78.

dedent()
Decreases the indentation.

getvalue()
Returns the buffer contents.

indent()
Increases the indentation.

indentation()
A context manager that increases the indentation.

section(*name*)
Helpful context manager that writes a paragraph, a heading, and the indents.

Parameters **name** – the section name that is written as heading.

write(*string*)
Writes a unicode string into the internal buffer.

`write_dl`(*rows*, *col_max=30*, *col_spacing=2*)
Writes a definition list into the buffer. This is how options and commands are usually formatted.

Parameters

- **`rows`** – a list of two item tuples for the terms and values.
- **`col_max`** – the maximum width of the first column.
- **`col_spacing`** – the number of spaces between the first and second column.

`write_heading`(*heading*)
Writes a heading into the buffer.

`write_paragraph`()
Writes a paragraph into the buffer.

`write_text`(*text*)
Writes re-indented text into the buffer. This rewraps and preserves paragraphs.

`write_usage`(*prog*, *args=''*, *prefix='Usage: '*)
Writes a usage line into the buffer.

Parameters

- **`prog`** – the program name.
- **`args`** – whitespace separated list of arguments.
- **`prefix`** – the prefix for the first line.

class `rudiments.reamed.click.`**`IntRange`**(*min=None*, *max=None*, *clamp=False*)
Bases: `click.types.IntParamType`

A parameter that works similar to `click.INT` but restricts the value to fit into a range. The default behavior is to fail if the value falls outside the range, but it can also be silently clamped between the two edges.

See ranges for an example.

`convert`(*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

`name = 'integer range'`

exception `rudiments.reamed.click.`**`MissingParameter`**(*message=None*, *ctx=None*, *param=None*, *param_hint=None*, *param_type=None*)
Bases: `click.exceptions.BadParameter`

Raised if click required an option or argument but it was not provided when invoking the script.

New in version 4.0.

Parameters **`param_type`** – a string that indicates the type of the parameter. The default is to inherit the parameter type from the given *param*. Valid values are `'parameter'`, `'option'` or `'argument'`.

`format_message`()

class `rudiments.reamed.click.`**`MultiCommand`**(*name=None*, *invoke_without_command=False*, *no_args_is_help=None*, *subcommand_metavar=None*, *chain=False*, *result_callback=None*, ***attrs*)
Bases: `click.core.Command`

A multi command is the basic implementation of a command that dispatches to subcommands. The most common version is the `Group`.

Parameters

- **`invoke_without_command`** – this controls how the multi command itself is invoked. By default it's only invoked if a subcommand is provided.
- **`no_args_is_help`** – this controls what happens if no arguments are provided. This option is enabled by default if *invoke_without_command* is disabled or disabled if it's enabled. If enabled this will add `--help` as argument if no arguments are passed.
- **`subcommand_metavar`** – the string that is used in the documentation to indicate the subcommand place.
- **`chain`** – if this is set to *True* chaining of multiple subcommands is enabled. This restricts the form of commands in that they cannot have optional arguments but it allows multiple commands to be chained together.
- **`result_callback`** – the result callback to attach to this multi command.

`allow_extra_args = True`

`allow_interspersed_args = False`

`collect_usage_pieces`(*ctx*)
Returns all the pieces that go into the usage line and returns it as a list of strings.

`format_commands`(*ctx*, *formatter*)
Extra format methods for multi methods that adds all the commands after the options.

`format_options`(*ctx*, *formatter*)
Writes all the options into the formatter if they exist.

`get_command`(*ctx*, *cmd_name*)
Given a context and a command name, this returns a `Command` object if it exists or returns *None*.

`invoke`(*ctx*)
Given a context, this invokes the attached callback (if it exists) in the right way.

`list_commands`(*ctx*)
Returns a list of subcommand names in the order they should appear.

`parse_args`(*ctx*, *args*)
Given a context and a list of arguments this creates the parser and parses the arguments, then modifies the context as necessary. This is automatically invoked by `make_context()`.

`resolve_command`(*ctx*, *args*)

`result_callback = None`
The result callback that is stored. This can be set or overridden with the `resultcallback()` decorator.

`resultcallback`(*replace=False*)
Adds a result callback to the chain command. By default if a result callback is already registered this will chain them but this can be disabled with the *replace* parameter. The result callback is invoked with the return value of the subcommand (or the list of return values from all subcommands if chaining is enabled) as well as the parameters as they would be passed to the main callback.

Example:

```
@click.group()
@click.option('-i', '--input', default=23)
def cli(input):
```

(continues on next page)

(continued from previous page)

```
    return 42

@cli.resultcallback()
def process_result(result, input):
    return result + input
```

New in version 3.0.

Parameters **replace** – if set to *True* an already existing result callback will be removed.

exception rudiments.reamed.click.**NoSuchOption**(*option_name*, *message=None*, *possibilities=None*, *ctx=None*)

Bases: click.exceptions.UsageError

Raised if click attempted to handle an option that does not exist.

New in version 4.0.

format_message()

class rudiments.reamed.click.**Option**(*param_decls=None*, *show_default=False*, *prompt=False*, *confirmation_prompt=False*, *hide_input=False*, *is_flag=None*, *flag_value=None*, *multiple=False*, *count=False*, *allow_from_autoenv=True*, *type=None*, *help=None*, *hidden=False*, *show_choices=True*, *show_envvar=False*, ***attrs*)

Bases: click.core.Parameter

Options are usually optional values on the command line and have some extra features that arguments don't have.

All other parameters are passed onwards to the parameter constructor.

Parameters

- **show_default** – controls if the default value should be shown on the help page. Normally, defaults are not shown. If this value is a string, it shows the string instead of the value. This is particularly useful for dynamic options.
- **show_envvar** – controls if an environment variable should be shown on the help page. Normally, environment variables are not shown.
- **prompt** – if set to *True* or a non empty string then the user will be prompted for input. If set to *True* the prompt will be the option name capitalized.
- **confirmation_prompt** – if set then the value will need to be confirmed if it was prompted for.
- **hide_input** – if this is *True* then the input on the prompt will be hidden from the user. This is useful for password input.
- **is_flag** – forces this option to act as a flag. The default is auto detection.
- **flag_value** – which value should be used for this flag if it's enabled. This is set to a boolean automatically if the option string contains a slash to mark two options.
- **multiple** – if this is set to *True* then the argument is accepted multiple times and recorded. This is similar to nargs in how it works but supports arbitrary number of arguments.
- **count** – this flag makes an option increment an integer.
- **allow_from_autoenv** – if this is enabled then the value of this parameter will be pulled from an environment variable in case a prefix is defined on the context.

- **help** – the help string.
- **hidden** – hide this option from help outputs.

add_to_parser (*parser*, *ctx*)

full_process_value (*ctx*, *value*)

get_default (*ctx*)
Given a context variable this calculates the default value.

get_help_record (*ctx*)

param_type_name = 'option'

prompt_for_value (*ctx*)
This is an alternative flow that can be activated in the full value processing if a value does not exist. It will prompt the user until a valid value exists and then returns the processed value as result.

resolve_envvar_value (*ctx*)

value_from_envvar (*ctx*)

class `rudiments.reamed.click.`**OptionParser** (*ctx=None*)
Bases: `object`

The option parser is an internal class that is ultimately used to parse options and arguments. It's modelled after optparse and brings a similar but vastly simplified API. It should generally not be used directly as the high level Click classes wrap it for you.

It's not nearly as extensible as optparse or argparse as it does not implement features that are implemented on a higher level (such as types or defaults).

Parameters **ctx** – optionally the `Context` where this parser should go with.

add_argument (*dest*, *nargs=1*, *obj=None*)
Adds a positional argument named *dest* to the parser.

The *obj* can be used to identify the option in the order list that is returned from the parser.

add_option (*opts*, *dest*, *action=None*, *nargs=1*, *const=None*, *obj=None*)
Adds a new option named *dest* to the parser. The destination is not inferred (unlike with optparse) and needs to be explicitly provided. Action can be any of `store`, `store_const`, `append`, `appnd_const` or `count`.

The *obj* can be used to identify the option in the order list that is returned from the parser.

allow_interspersed_args = None
This controls how the parser deals with interspersed arguments. If this is set to *False*, the parser will stop on the first non-option. Click uses this to implement nested subcommands safely.

ctx = None
The `Context` for this parser. This might be *None* for some advanced use cases.

ignore_unknown_options = None
This tells the parser how to deal with unknown options. By default it will error out (which is sensible), but there is a second mode where it will ignore it and continue processing after shifting all the unknown options into the resulting args.

parse_args (*args*)
Parses positional arguments and returns `(values, args, order)` for the parsed options and arguments as well as the leftover arguments if there are any. The order is a list of objects as they appear on the command line. If arguments appear multiple times they will be memorized multiple times as well.

class rudiments.reamed.click.**ParamType**

Bases: `object`

Helper for converting values through types. The following is necessary for a valid type:

- it needs a name
- it needs to pass through None unchanged
- it needs to convert from a string
- it needs to convert its result type through unchanged (eg: needs to be idempotent)
- it needs to be able to deal with param and context being *None*. This can be the case when the object is used with prompt inputs.

convert(*value*, *param*, *ctx*)

Converts the value. This is not invoked for values that are *None* (the missing value).

envvar_list_splitter = None

if a list of this type is expected and the value is pulled from a string environment variable, this is what splits it up. *None* means any whitespace. For all parameters the general rule is that whitespace splits them up. The exception are paths and files which are split by `os.path.pathsep` by default (":" on Unix and ";" on Windows).

fail(*message*, *param=None*, *ctx=None*)

Helper method to fail with an invalid value message.

get_metavar(*param*)

Returns the metavar default for this param if it provides one.

get_missing_message(*param*)

Optionally might return extra information about a missing parameter.

New in version 2.0.

is_composite = False

name = None

the descriptive name of this type

split_envvar_value(*rv*)

Given a value from an environment variable this splits it up into small chunks depending on the defined envvar list splitter.

If the splitter is set to *None*, which means that whitespace splits, then leading and trailing whitespace is ignored. Otherwise, leading and trailing splitters usually lead to empty items being included.

class rudiments.reamed.click.**Parameter**(*param_decls=None*, *type=None*, *required=False*, *default=None*, *callback=None*, *nargs=None*, *metavar=None*, *expose_value=True*, *is_eager=False*, *envvar=None*, *autocompletion=None*)

Bases: `object`

A parameter to a command comes in two versions: they are either *Option*s or *Argument*s. Other subclasses are currently not supported by design as some of the internals for parsing are intentionally not finalized.

Some settings are supported by both options and arguments.

Parameters

- **param_decls** – the parameter declarations for this option or argument. This is a list of flags or argument names.

- **type** – the type that should be used. Either a *ParamType* or a Python type. The later is converted into the former automatically if supported.
- **required** – controls if this is optional or not.
- **default** – the default value if omitted. This can also be a callable, in which case it's invoked when the default is needed without any arguments.
- **callback** – a callback that should be executed after the parameter was matched. This is called as `fn(ctx, param, value)` and needs to return the value.
- **nargs** – the number of arguments to match. If not `1` the return value is a tuple instead of single value. The default for nargs is `1` (except if the type is a tuple, then it's the arity of the tuple).
- **metavar** – how the value is represented in the help page.
- **expose_value** – if this is *True* then the value is passed onwards to the command callback and stored on the context, otherwise it's skipped.
- **is_eager** – eager values are processed before non eager ones. This should not be set for arguments or it will inverse the order of processing.
- **envvar** – a string or list of strings that are environment variables that should be checked.

Changed in version 7.1: Empty environment variables are ignored rather than taking the empty string value. This makes it possible for scripts to clear variables if they can't unset them.

Changed in version 2.0: Changed signature for parameter callback to also be passed the parameter. The old callback format will still work, but it will raise a warning to give you a chance to migrate the code easier.

add_to_parser(*parser*, *ctx*)

consume_value(*ctx*, *opts*)

full_process_value(*ctx*, *value*)

get_default(*ctx*)
Given a context variable this calculates the default value.

get_error_hint(*ctx*)
Get a stringified version of the param for use in error messages to indicate which param caused the error.

get_help_record(*ctx*)

get_usage_pieces(*ctx*)

handle_parse_result(*ctx*, *opts*, *args*)

property human_readable_name
Returns the human readable name of this parameter. This is the same as the name for options, but the metavar for arguments.

make_metavar()

param_type_name = 'parameter'

process_value(*ctx*, *value*)
Given a value and context this runs the logic to convert the value as necessary.

resolve_envvar_value(*ctx*)

type_cast_value(*ctx*, *value*)
Given a value this runs it properly through the type system. This automatically handles things like *nargs* and *multiple* as well as composite types.

value_from_envvar(*ctx*)

value_is_missing(*value*)

class rudiments.reamed.click.**Path**(*exists=False*, *file_okay=True*, *dir_okay=True*, *writable=False*, *readable=True*, *resolve_path=False*, *allow_dash=False*, *path_type=None*)

Bases: `click.types.ParamType`

The path type is similar to the *File* type but it performs different checks. First of all, instead of returning an open file handle it returns just the filename. Secondly, it can perform various basic checks about what the file or directory should be.

Changed in version 6.0: *allow_dash* was added.

Parameters

- **exists** – if set to true, the file or directory needs to exist for this value to be valid. If this is not required and a file does indeed not exist, then all further checks are silently skipped.
- **file_okay** – controls if a file is a possible value.
- **dir_okay** – controls if a directory is a possible value.
- **writable** – if true, a writable check is performed.
- **readable** – if true, a readable check is performed.
- **resolve_path** – if this is true, then the path is fully resolved before the value is passed onwards. This means that it's absolute and symlinks are resolved. It will not expand a tilde-prefix, as this is supposed to be done by the shell only.
- **allow_dash** – If this is set to *True*, a single dash to indicate standard streams is permitted.
- **path_type** – optionally a string type that should be used to represent the path. The default is *None* which means the return value will be either bytes or unicode depending on what makes most sense given the input data Click deals with.

coerce_path_result(*rv*)

convert(*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

envvar_list_splitter = ':'

class rudiments.reamed.click.**Tuple**(*types*)

Bases: `click.types.CompositeParamType`

The default behavior of Click is to apply a type on a value directly. This works well in most cases, except for when *nargs* is set to a fixed count and different types should be used for different items. In this case the *Tuple* type can be used. This type can only be used if *nargs* is set to a fixed number.

For more information see tuple-type.

This can be selected by using a Python tuple literal as a type.

Parameters **types** – a list of types that should be used for the tuple items.

property arity

convert(*value*, *param*, *ctx*)
Converts the value. This is not invoked for values that are *None* (the missing value).

property name

exception `rudiments.reamed.click.`**`UsageError`**(*message*, *ctx=None*)

Bases: `click.exceptions.ClickException`

An internal exception that signals a usage error. This typically aborts any further handling.

Parameters

- **`message`** – the error message to display.
- **`ctx`** – optionally the context that caused this error. Click will fill in the context automatically in some situations.

`exit_code = 2`

`show`(*file=None*)

`rudiments.reamed.click.`**`argument`**(**param_decls*, ***attrs*)

Attaches an argument to the command. All positional arguments are passed as parameter declarations to *`Argument`*; all keyword arguments are forwarded unchanged (except `cls`). This is equivalent to creating an *`Argument`* instance manually and attaching it to the *`Command.params`* list.

Parameters **`cls`** – the argument class to instantiate. This defaults to *`Argument`*.

`rudiments.reamed.click.`**`clear`**()

Clears the terminal screen. This will have the effect of clearing the whole visible space of the terminal and moving the cursor to the top left. This does not do anything if not connected to a terminal.

New in version 2.0.

`rudiments.reamed.click.`**`command`**(*name=None*, *cls=None*, ***attrs*)

Creates a new *`Command`* and uses the decorated function as callback. This will also automatically attach all decorated *`option()`*s and *`argument()`*s as parameters to the command.

The name of the command defaults to the name of the function with underscores replaced by dashes. If you want to change that, you can pass the intended name as the first argument.

All keyword arguments are forwarded to the underlying command class.

Once decorated the function turns into a *`Command`* instance that can be invoked as a command line utility or be attached to a command *`Group`*.

Parameters

- **`name`** – the name of the command. This defaults to the function name with underscores replaced by dashes.
- **`cls`** – the command class to instantiate. This defaults to *`Command`*.

`rudiments.reamed.click.`**`confirm`**(*text*, *default=False*, *abort=False*, *prompt_suffix=': '*, *show_default=True*, *err=False*)

Prompts for confirmation (yes/no question).

If the user aborts the input by sending a interrupt signal this function will catch it and raise a *`Abort`* exception.

New in version 4.0: Added the *err* parameter.

Parameters

- **`text`** – the question to ask.
- **`default`** – the default for the prompt.
- **`abort`** – if this is set to *True* a negative answer aborts the exception by raising *`Abort`*.
- **`prompt_suffix`** – a suffix that should be added to the prompt.
- **`show_default`** – shows or hides the default value in the prompt.

- **err** – if set to true the file defaults to `stderr` instead of `stdout`, the same as with echo.

`rudiments.reamed.click.`**confirmation_option**(**param_decls*, ***attrs*)

Shortcut for confirmation prompts that can be ignored by passing `--yes` as parameter.

This is equivalent to decorating a function with `option()` with the following parameters:

```
def callback(ctx, param, value):
    if not value:
        ctx.abort()

@click.command()
@click.option('--yes', is_flag=True, callback=callback,
              expose_value=False, prompt='Do you want to continue?')
def dropdb():
    pass
```

`rudiments.reamed.click.`**echo**(*message=None*, *file=None*, *nl=True*, *err=False*, *color=None*)

Prints a message plus a newline to the given file or stdout. On first sight, this looks like the print function, but it has improved support for handling Unicode and binary data that does not fail no matter how badly configured the system is.

Primarily it means that you can print binary data as well as Unicode data on both 2.x and 3.x to the given file in the most appropriate way possible. This is a very carefree function in that it will try its best to not fail. As of Click 6.0 this includes support for unicode output on the Windows console.

In addition to that, if colorama is installed, the echo function will also support clever handling of ANSI codes. Essentially it will then do the following:

- add transparent handling of ANSI color codes on Windows.
- hide ANSI codes automatically if the destination file is not a terminal.

Changed in version 6.0: As of Click 6.0 the echo function will properly support unicode output on the windows console. Not that click does not modify the interpreter in any way which means that *sys.stdout* or the print statement or function will still not provide unicode support.

Changed in version 2.0: Starting with version 2.0 of Click, the echo function will work with colorama if it's installed.

New in version 3.0: The *err* parameter was added.

Changed in version 4.0: Added the *color* flag.

Parameters

- **message** – the message to print
- **file** – the file to write to (defaults to `stdout`)
- **err** – if set to true the file defaults to `stderr` instead of `stdout`. This is faster and easier than calling `get_text_stderr()` yourself.
- **nl** – if set to *True* (the default) a newline is printed afterwards.
- **color** – controls if the terminal supports ANSI colors or not. The default is autodetection.

`rudiments.reamed.click.`**echo_via_pager**(*text_or_generator*, *color=None*)

This function takes a text and shows it via an environment specific pager on stdout.

Changed in version 3.0: Added the *color* flag.

Parameters

- **`text_or_generator`** – the text to page, or alternatively, a generator emitting the text to page.
- **`color`** – controls if the pager supports ANSI colors or not. The default is autodetection.

`rudiments.reamed.click.`**`edit`**(*text=None*, *editor=None*, *env=None*, *require_save=True*, *extension='.txt'*, *filename=None*)

Edits the given text in the defined editor. If an editor is given (should be the full path to the executable but the regular operating system search path is used for finding the executable) it overrides the detected editor. Optionally, some environment variables can be used. If the editor is closed without changes, *None* is returned. In case a file is edited directly the return value is always *None* and *require_save* and *extension* are ignored.

If the editor cannot be opened a *`UsageError`* is raised.

Note for Windows: to simplify cross-platform usage, the newlines are automatically converted from POSIX to Windows and vice versa. As such, the message here will have `\n` as newline markers.

Parameters

- **`text`** – the text to edit.
- **`editor`** – optionally the editor to use. Defaults to automatic detection.
- **`env`** – environment variables to forward to the editor.
- **`require_save`** – if this is true, then not saving in the editor will make the return value become *None*.
- **`extension`** – the extension to tell the editor about. This defaults to *.txt* but changing this might change syntax highlighting.
- **`filename`** – if provided it will edit this file instead of the provided text contents. It will not use a temporary file as an indirection in that case.

`rudiments.reamed.click.`**`format_filename`**(*filename*, *shorten=False*)

Formats a filename for user display. The main purpose of this function is to ensure that the filename can be displayed at all. This will decode the filename to unicode if necessary in a way that it will not fail. Optionally, it can shorten the filename to not include the full path to the filename.

Parameters

- **`filename`** – formats a filename for UI display. This will also convert the filename into unicode without failing.
- **`shorten`** – this optionally shortens the filename to strip of the path that leads up to it.

`rudiments.reamed.click.`**`get_app_dir`**(*app_name*, *roaming=True*, *force_posix=False*)

Returns the config folder for the application. The default behavior is to return whatever is most appropriate for the operating system.

To give you an idea, for an app called `"Foo Bar"`, something like the following folders could be returned:

Mac OS X: `~/Library/Application Support/Foo Bar`

Mac OS X (POSIX): `~/.foo-bar`

Unix: `~/.config/foo-bar`

Unix (POSIX): `~/.foo-bar`

Win XP (roaming): `C:\Documents and Settings\<user>\Local Settings\Application Data\Foo Bar`

Win XP (not roaming): `C:\Documents and Settings\<user>\Application Data\Foo Bar`

Win 7 (roaming): `C:\Users\<user>\AppData\Roaming\Foo Bar`

Win 7 (not roaming): `C:\Users\<user>\AppData\Local\Foo Bar`

New in version 2.0.

Parameters

- **`app_name`** – the application name. This should be properly capitalized and can contain whitespace.
- **`roaming`** – controls if the folder should be roaming or not on Windows. Has no affect otherwise.
- **`force_posix`** – if this is set to *True* then on any POSIX system the folder will be stored in the home folder with a leading dot instead of the XDG config home or darwin's application support folder.

`rudiments.reamed.click.`**`get_binary_stream`**`(`*name*`)`

Returns a system stream for byte processing. This essentially returns the stream from the sys module with the given name but it solves some compatibility issues between different Python versions. Primarily this function is necessary for getting binary streams on Python 3.

Parameters **`name`** – the name of the stream to open. Valid names are `'stdin'`, `'stdout'` and `'stderr'`

`rudiments.reamed.click.`**`get_current_context`**`(`*silent=False*`)`

Returns the current click context. This can be used as a way to access the current context object from anywhere. This is a more implicit alternative to the *`pass_context()`* decorator. This function is primarily useful for helpers such as *`echo()`* which might be interested in changing its behavior based on the current context.

To push the current context, *`Context.scope()`* can be used.

New in version 5.0.

Parameters **`silent`** – if set to *True* the return value is *None* if no context is available. The default behavior is to raise a `RuntimeError`.

`rudiments.reamed.click.`**`get_os_args`**`()`

This returns the argument part of sys.argv in the most appropriate form for processing. What this means is that this return value is in a format that works for Click to process but does not necessarily correspond well to what's actually standard for the interpreter.

On most environments the return value is `sys.argv[:1]` unchanged. However if you are on Windows and running Python 2 the return value will actually be a list of unicode strings instead because the default behavior on that platform otherwise will not be able to carry all possible values that sys.argv can have.

New in version 6.0.

`rudiments.reamed.click.`**`get_terminal_size`**`()`

Returns the current size of the terminal as tuple in the form `(width, height)` in columns and rows.

`rudiments.reamed.click.`**`get_text_stream`**`(`*name*, *encoding=None*, *errors='strict'*`)`

Returns a system stream for text processing. This usually returns a wrapped stream around a binary stream returned from *`get_binary_stream()`* but it also can take shortcuts on Python 3 for already correctly configured streams.

Parameters

- **`name`** – the name of the stream to open. Valid names are `'stdin'`, `'stdout'` and `'stderr'`
- **`encoding`** – overrides the detected default encoding.

- **errors** – overrides the default error mode.

`rudiments.reamed.click.`**`getchar`**`(`*echo=False*`)`

Fetches a single character from the terminal and returns it. This will always return a unicode character and under certain rare circumstances this might return more than one character. The situations which more than one character is returned is when for whatever reason multiple characters end up in the terminal buffer or standard input was not actually a terminal.

Note that this will always read from the terminal, even if something is piped into the standard input.

Note for Windows: in rare cases when typing non-ASCII characters, this function might wait for a second character and then return both at once. This is because certain Unicode characters look like special-key markers.

New in version 2.0.

Parameters **echo** – if set to *True*, the character read will also show up on the terminal. The default is to not show it.

`rudiments.reamed.click.`**`group`**`(`*name=None*, ***attrs*`)`

Creates a new *`Group`* with a function as callback. This works otherwise the same as *`command()`* just that the *cls* parameter is set to *`Group`*.

`rudiments.reamed.click.`**`help_option`**`(`**param_decls*, ***attrs*`)`

Adds a `--help` option which immediately ends the program printing out the help page. This is usually unnecessary to add as this is added by default to all commands unless suppressed.

Like *`version_option()`*, this is implemented as eager option that prints in the callback and exits.

All arguments are forwarded to *`option()`*.

`rudiments.reamed.click.`**`launch`**`(`*url*, *wait=False*, *locate=False*`)`

This function launches the given URL (or filename) in the default viewer application for this file type. If this is an executable, it might launch the executable in a new session. The return value is the exit code of the launched application. Usually, `0` indicates success.

Examples:

```
click.launch('https://click.palletsprojects.com/')
click.launch('/my/downloaded/file', locate=True)
```

New in version 2.0.

Parameters

- **url** – URL or filename of the thing to launch.
- **wait** – waits for the program to stop.
- **locate** – if this is set to *True* then instead of launching the application associated with the URL it will attempt to launch a file manager with the file located. This might have weird effects if the URL does not point to the filesystem.

`rudiments.reamed.click.`**`make_pass_decorator`**`(`*object_type*, *ensure=False*`)`

Given an object type this creates a decorator that will work similar to *`pass_obj()`* but instead of passing the object of the current context, it will find the innermost context of type `object_type()`.

This generates a decorator that works roughly like this:

```
from functools import update_wrapper

def decorator(f):
    @pass_context
    def new_func(ctx, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        obj = ctx.find_object(object_type)
        return ctx.invoke(f, obj, *args, **kwargs)
    return update_wrapper(new_func, f)
return decorator
```

Parameters

- **`object_type`** – the type of the object to pass.
- **`ensure`** – if set to *True*, a new object will be created and remembered on the context if it's not there yet.

`rudiments.reamed.click.`**`open_file`**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *lazy=False*, *atomic=False*)

This is similar to how the *`File`* works but for manual usage. Files are opened non lazy by default. This can open regular files as well as stdin/stdout if `'-'` is passed.

If stdin/stdout is returned the stream is wrapped so that the context manager will not close the stream accidentally. This makes it possible to always use the function like this without having to worry to accidentally close a standard stream:

```
with open_file(filename) as f:
    ...
```

New in version 3.0.

Parameters

- **`filename`** – the name of the file to open (or `'-'` for stdin/stdout).
- **`mode`** – the mode in which to open the file.
- **`encoding`** – the encoding to use.
- **`errors`** – the error handling for this file.
- **`lazy`** – can be flipped to true to open the file lazily.
- **`atomic`** – in atomic mode writes go into a temporary file and it's moved on close.

`rudiments.reamed.click.`**`option`**(**param_decls*, ***attrs*)

Attaches an option to the command. All positional arguments are passed as parameter declarations to *`Option`*; all keyword arguments are forwarded unchanged (except `cls`). This is equivalent to creating an *`Option`* instance manually and attaching it to the *`Command.params`* list.

Parameters **`cls`** – the option class to instantiate. This defaults to *`Option`*.

`rudiments.reamed.click.`**`pass_context`**(*f*)

Marks a callback as wanting to receive the current context object as first argument.

`rudiments.reamed.click.`**`pass_obj`**(*f*)

Similar to *`pass_context()`*, but only pass the object on the context onwards (*`Context.obj`*). This is useful if that object represents the state of a nested system.

`rudiments.reamed.click.`**`password_option`**(**param_decls*, ***attrs*)

Shortcut for password prompts.

This is equivalent to decorating a function with *`option()`* with the following parameters:

```
@click.command()
@click.option('--password', prompt=True, confirmation_prompt=True,
              hide_input=True)
def changeadmin(password):
    pass
```

`rudiments.reamed.click.`**`pause`**(*info='Press any key to continue ...'*, *err=False*)

This command stops execution and waits for the user to press any key to continue. This is similar to the Windows batch "pause" command. If the program is not run through a terminal, this command will instead do nothing.

New in version 2.0.

New in version 4.0: Added the *err* parameter.

Parameters

- **`info`** – the info string to print before pausing.
- **`err`** – if set to message goes to `stderr` instead of `stdout`, the same as with echo.

`rudiments.reamed.click.`**`progressbar`**(*iterable=None*, *length=None*, *label=None*, *show_eta=True*, *show_percent=None*, *show_pos=False*, *item_show_func=None*, *fill_char='#'*, *empty_char='-'*, *bar_template='%(label)s [%(bar)s] %(info)s'*, *info_sep=' '*, *width=36*, *file=None*, *color=None*)

This function creates an iterable context manager that can be used to iterate over something while showing a progress bar. It will either iterate over the *iterable* or *length* items (that are counted up). While iteration happens, this function will print a rendered progress bar to the given *file* (defaults to stdout) and will attempt to calculate remaining time and more. By default, this progress bar will not be rendered if the file is not a terminal.

The context manager creates the progress bar. When the context manager is entered the progress bar is already created. With every iteration over the progress bar, the iterable passed to the bar is advanced and the bar is updated. When the context manager exits, a newline is printed and the progress bar is finalized on screen.

Note: The progress bar is currently designed for use cases where the total progress can be expected to take at least several seconds. Because of this, the ProgressBar class object won't display progress that is considered too fast, and progress where the time between steps is less than a second.

No printing must happen or the progress bar will be unintentionally destroyed.

Example usage:

```
with progressbar(items) as bar:
    for item in bar:
        do_something_with(item)
```

Alternatively, if no iterable is specified, one can manually update the progress bar through the *update()* method instead of directly iterating over the progress bar. The update method accepts the number of steps to increment the bar with:

```
with progressbar(length=chunks.total_bytes) as bar:
    for chunk in chunks:
        process_chunk(chunk)
        bar.update(chunks.bytes)
```

New in version 2.0.

New in version 4.0: Added the *color* parameter. Added a *update* method to the progressbar object.

Parameters

- **`iterable`** – an iterable to iterate over. If not provided the length is required.
- **`length`** – the number of items to iterate over. By default the progressbar will attempt to ask the iterator about its length, which might or might not work. If an iterable is also provided this parameter can be used to override the length. If an iterable is not provided the progress bar will iterate over a range of that length.
- **`label`** – the label to show next to the progress bar.
- **`show_eta`** – enables or disables the estimated time display. This is automatically disabled if the length cannot be determined.
- **`show_percent`** – enables or disables the percentage display. The default is *True* if the iterable has a length or *False* if not.
- **`show_pos`** – enables or disables the absolute position display. The default is *False*.
- **`item_show_func`** – a function called with the current item which can return a string to show the current item next to the progress bar. Note that the current item can be *None*!
- **`fill_char`** – the character to use to show the filled part of the progress bar.
- **`empty_char`** – the character to use to show the non-filled part of the progress bar.
- **`bar_template`** – the format string to use as template for the bar. The parameters in it are `label` for the label, `bar` for the progress bar and `info` for the info section.
- **`info_sep`** – the separator between multiple info items (eta etc.)
- **`width`** – the width of the progress bar in characters, 0 means full terminal width
- **`file`** – the file to write to. If this is not a terminal then only the label is printed.
- **`color`** – controls if the terminal supports ANSI colors or not. The default is autodetection. This is only needed if ANSI codes are included anywhere in the progress bar output which is not the case by default.

`rudiments.reamed.click.`**`prompt`**(*text*, *default=None*, *hide_input=False*, *confirmation_prompt=False*, *type=None*, *value_proc=None*, *prompt_suffix=': '*, *show_default=True*, *err=False*, *show_choices=True*)

Prompts a user for input. This is a convenience function that can be used to prompt a user for input later.

If the user aborts the input by sending a interrupt signal, this function will catch it and raise a `Abort` exception.

New in version 7.0: Added the show_choices parameter.

New in version 6.0: Added unicode support for cmd.exe on Windows.

New in version 4.0: Added the *err* parameter.

Parameters

- **`text`** – the text to show for the prompt.
- **`default`** – the default value to use if no input happens. If this is not given it will prompt until it's aborted.
- **`hide_input`** – if this is set to true then the input value will be hidden.
- **`confirmation_prompt`** – asks for confirmation for the value.
- **`type`** – the type to use to check the value against.
- **`value_proc`** – if this parameter is provided it's a function that is invoked instead of the type conversion to convert a value.

- **prompt_suffix** – a suffix that should be added to the prompt.
- **show_default** – shows or hides the default value in the prompt.
- **err** – if set to true the file defaults to `stderr` instead of `stdout`, the same as with echo.
- **show_choices** – Show or hide choices if the passed type is a Choice. For example if type is a Choice of either day or week, show_choices is true and text is "Group by" then the prompt will be "Group by (day, week): ".

`rudiments.reamed.click.`**`secho`**(*message=None*, *file=None*, *nl=True*, *err=False*, *color=None*, ***styles*)

This function combines `echo()` and `style()` into one call. As such the following two calls are the same:

```
click.secho('Hello World!', fg='green')
click.echo(click.style('Hello World!', fg='green'))
```

All keyword arguments are forwarded to the underlying functions depending on which one they go with.

New in version 2.0.

`rudiments.reamed.click.`**`style`**(*text*, *fg=None*, *bg=None*, *bold=None*, *dim=None*, *underline=None*, *blink=None*, *reverse=None*, *reset=True*)

Styles a text with ANSI styles and returns the new string. By default the styling is self contained which means that at the end of the string a reset code is issued. This can be prevented by passing `reset=False`.

Examples:

```
click.echo(click.style('Hello World!', fg='green'))
click.echo(click.style('ATTENTION!', blink=True))
click.echo(click.style('Some things', reverse=True, fg='cyan'))
```

Supported color names:

- `black` (might be a gray)
- `red`
- `green`
- `yellow` (might be an orange)
- `blue`
- `magenta`
- `cyan`
- `white` (might be light gray)
- `bright_black`
- `bright_red`
- `bright_green`
- `bright_yellow`
- `bright_blue`
- `bright_magenta`
- `bright_cyan`
- `bright_white`
- `reset` (reset the color code only)

New in version 2.0.

New in version 7.0: Added support for bright colors.

Parameters

- **`text`** – the string to style with ansi codes.
- **`fg`** – if provided this will become the foreground color.
- **`bg`** – if provided this will become the background color.
- **`bold`** – if provided this will enable or disable bold mode.
- **`dim`** – if provided this will enable or disable dim mode. This is badly supported.
- **`underline`** – if provided this will enable or disable underline.
- **`blink`** – if provided this will enable or disable blinking.
- **`reverse`** – if provided this will enable or disable inverse rendering (foreground becomes background and the other way round).
- **`reset`** – by default a reset-all code is added at the end of the string which means that styles do not carry over. This can be disabled to compose styles.

`rudiments.reamed.click.`**`unstyle`**`(`*text*`)`

Removes ANSI styling information from a string. Usually it's not necessary to use this function as Click's echo function will automatically remove styling if necessary.

New in version 2.0.

Parameters **`text`** – the text to remove style information from.

`rudiments.reamed.click.`**`version_option`**`(`*version=None*, **param_decls*, ***attrs*`)`

Adds a `--version` option which immediately ends the program printing out the version number. This is implemented as an eager option that prints the version and exits the program in the callback.

Parameters

- **`version`** – the version number to show. If not provided Click attempts an auto discovery via setuptools.
- **`prog_name`** – the name of the program (defaults to autodetection)
- **`message`** – custom message to show instead of the default (`'%(prog)s, version %(version)s'`)
- **`others`** – everything else is forwarded to `option()`.

`rudiments.reamed.click.`**`wrap_text`**`(`*text*, *width=78*, *initial_indent=''*, *subsequent_indent=''*, *preserve_paragraphs=False*`)`

A helper function that intelligently wraps text. By default, it assumes that it operates on a single paragraph of text but if the *preserve_paragraphs* parameter is provided it will intelligently handle paragraphs (defined by two empty lines).

If paragraphs are handled, a paragraph can be prefixed with an empty line containing the `\b` character (`\x08`) to indicate that no rewrapping should happen in that block.

Parameters

- **`text`** – the text that should be rewrapped.
- **`width`** – the maximum width for the text.
- **`initial_indent`** – the initial indent that should be placed on the first line as a string.
- **`subsequent_indent`** – the indent string that should be placed on each consecutive line.

- **preserve_paragraphs** – if this flag is set then the wrapping will intelligently handle paragraphs.

`rudiments.reamed.click.`**`pretty_path`**(*path*, *_home_re=re.compile('^/home/docs/')*)
Prettify path for humans, and make it Unicode.

`rudiments.reamed.click.`**`serror`**(*message*, **args*, ***kwargs*)
Print a styled error message, while using any arguments to format the message.

exception `rudiments.reamed.click.`**`LoggedFailure`**(*message*)
Bases: `click.exceptions.UsageError`

Report a failure condition to the user.

class `rudiments.reamed.click.`**`AliasedGroup`**(*name=None*, *commands=None*, ***attrs*)
Bases: `click.core.Group`

A command group with alias names.

Inherit from this class and define a `MAP` class variable, which is a mapping from alias names to canonical command names. Then use that derived class as the `cls` parameter for a `click.group` decorator.

`MAP = {}`

`get_command`(*ctx*, *cmd_name*)
Map some aliases to their 'real' names.

class `rudiments.reamed.click.`**`Configuration`**(*name*, *config_paths=None*, *project=None*)
Bases: `object`

Configuration container that is initialized early in the main command.

The default instance is available via the Click context as `ctx.obj.cfg`. Configuration is lazily loaded, on first access.

`DEFAULT_CONFIG_OPTS = {'default_encoding': 'utf-8', 'encoding': 'utf-8'}`

`DEFAULT_PATH = ['/etc/{appname}.conf', '/etc/{appname}.d/', '{appcfg}.conf']`

`NO_DEFAULT = <object object>`

`dump`(*to=None*)
Dump the merged configuration to a stream or stdout.

classmethod **`from_context`**(*ctx*, *config_paths=None*, *project=None*)
Create a configuration object, and initialize the Click context with it.

`get`(*name*, *default=<object object>*)
Return the specified name from the root section.

Parameters

- **name** (*str*) – The name of the requested value.
- **default** (*optional*) – If set, the default value to use instead of raising *LoggedFailure* for unknown names.

Returns The value for *name*.

Raises ***LoggedFailure*** – The requested *name* was not found.

`load`()
Load configuration from the defined locations.

locations(*exists=True*)
Return the location of the config file(s).

A given directory will be scanned for `*.conf` files, in alphabetical order. Any duplicates will be eliminated.

If `exists` is True, only existing configuration locations are returned.

section(*ctx*, *optional=False*)
Return section of the config for a specific context (sub-command).

Parameters

- **ctx** (`Context`) – The Click context object.
- **optional** (*bool*) – If `True`, return an empty config object when section is missing.

Returns

The configuration section belonging to the active (sub-)command (based on `ctx.info_name`).

Return type Section

4.4.1.2 Submodules

4.4.1.3 rudiments.humanize module

I/O of common values in forms understood by humans.

`rudiments.humanize.`**bytes2iec**(*size*, *compact=False*)
Convert a size value in bytes to its equivalent in IEC notation.

See http://physics.nist.gov/cuu/Units/binary.html.

Parameters

- **size** (*int*) – Number of bytes.
- **compact** (*bool*) – If `True`, the result contains no spaces.

Returns String representation of `size`.

Raises ValueError – Negative or out of bounds value for `size`.

`rudiments.humanize.`**iec2bytes**(*size_spec*, *only_positive=True*)
Convert a size specification, optionally containing a scaling unit in IEC notation, to a number of bytes.

Parameters

- **size_spec** (*str*) – Number, optionally followed by a unit.
- **only_positive** (*bool*) – Allow only positive values?

Returns Numeric bytes size.

Raises ValueError – Unknown unit specifiers, or bad leading integer.

`rudiments.humanize.`**merge_adjacent**(*numbers*, *indicator='..'*, *base=0*)
Merge adjacent numbers in an iterable of numbers.

Parameters

- **numbers** (*list*) – List of integers or numeric strings.
- **indicator** (*str*) – Delimiter to indicate generated ranges.

- **base** (*int*) – Passed to the *int()* conversion when comparing numbers.

Returns Condensed sequence with either ranges or isolated numbers.

Return type list of str

4.4.1.4 rudiments.morph module

Data type conversions.

4.4.1.5 rudiments.pysupport module

Python helpers + magic.

`rudiments.pysupport.`**`import_name`**(*modulename*, *name=None*)

Import identifier `name` from module `modulename`.

If `name` is omitted, `modulename` must contain the name after the module path, delimited by a colon.

Parameters

- **modulename** (*str*) – Fully qualified module name, e.g. `x.y.z`.
- **name** (*str*) – Name to import from `modulename`.

Returns Requested object.

Return type object

`rudiments.pysupport.`**`load_module`**(*modulename*, *modulepath*)

Load a Python module from a path under a specified name.

Parameters

- **modulename** (*str*) – Fully qualified module name, e.g. `x.y.z`.
- **modulepath** (*str*) – Filename of the module.

Returns Loaded module.

4.4.1.6 rudiments.security module

Security / AuthN / AuthZ helpers.

class `rudiments.security.`**`Credentials`**(*target*)

Bases: `object`

Look up and provide authN credentials (username / password) from common sources.

`AUTH_MEMOIZE_INPUT = {}`

`NETRC_FILE = None`

`URL_RE = re.compile('^(http|https|ftp|ftps)://')`

`auth_pair`(*force_console=False*)

Return username/password tuple, possibly prompting the user for them.

`auth_valid`()

Return bool indicating whether full credentials were provided.

4.4.1.7 rudiments.system module

Operating system related stdlib extensions.

4.4.1.8 rudiments.www module

WWW access helpers.

You need a dependency on requests in your project if you use this module.

`rudiments.www.`**`url_as_file`**`(`*url*, *ext=None*`)`

Context manager that GETs a given *url* and provides it as a local file.

The file is in a closed state upon entering the context, and removed when leaving it, if still there.

To give the file name a specific extension, use *ext*; the extension can optionally include a separating dot, otherwise it will be added.

Parameters

- **`url`** (*`str`*) – URL to retrieve.
- **`ext`** (*`str, optional`*) – Extension for the generated filename.

Yields *str* – The path to a temporary file with the content of the URL.

Raises **`requests.RequestException`** – Base exception of `requests`, see its docs for more detailed ones.

Example

```
>>> import io, re, json
>>> with url_as_file('https://api.github.com/meta', ext='json') as meta:
...     meta, json.load(io.open(meta, encoding='ascii'))['hooks']
(u'/tmp/www-api.github.com-Ba5OhD.json', [u'192.30.252.0/22'])
```

4.5 Contribution Guidelines

4.5.1 Overview

Contributing to this project is easy, and reporting an issue or adding to the documentation also improves things for every user. You don’t need to be a developer to contribute.

4.5.1.1 Reporting issues

Please use the *GitHub issue tracker*, and describe your problem so that it can be easily reproduced. Providing relevant version information on the project itself and your environment helps with that.

4.5.1.2 Improving documentation

The easiest way to provide examples or related documentation that helps other users is the *GitHub wiki*.

If you are comfortable with the Sphinx documentation tool, you can also prepare a pull request with changes to the core documentation. GitHub's built-in text editor makes this especially easy, when you choose the *"Create a new branch for this commit and start a pull request"* option on saving. Small fixes for typos and the like are a matter of minutes when using that tool.

4.5.1.3 Code contributions

Here's a quick guide to improve the code:

1. Fork the repo, and clone the fork to your machine.
2. Add your improvements, the technical details are further below.
3. Run the tests and make sure they're passing (`invoke test`).
4. Check for violations of code conventions (`invoke check`).
5. Make sure the documentation builds without errors (`invoke build --docs`).
6. Push to your fork and submit a pull request.

Please be patient while waiting for a review. Life & work tend to interfere.

4.5.2 Details on contributing code

This project is written in Python, and the documentation is generated using Sphinx. setuptools and Invoke are used to build and manage the project. Tests are written and executed using pytest and tox.

4.5.2.1 Set up a working development environment

To set up a working directory from your own fork, follow these steps, but replace the repository `https` URLs with SSH ones that point to your fork.

For that to work on Debian type systems, you need the `git`, `python`, and `python-virtualenv` packages installed. Other distributions are similar.

4.5.2.2 Add your changes to a feature branch

For any cohesive set of changes, create a *new* branch based on the current upstream `master`, with a name reflecting the essence of your improvement.

```
git branch "name-for-my-fixes" origin/master
git checkout "name-for-my-fixes"
... make changes...
invoke ci # check output for broken tests, or PEP8 violations and the like
... commit changes...
git push origin "name-for-my-fixes"
```

Please don't create large lumps of unrelated changes in a single pull request. Also take extra care to avoid spurious changes, like mass whitespace diffs. All Python sources use spaces to indent, not TABs.

4.5.2.3 Make sure your changes work

Some things that will increase the chance that your pull request is accepted:

- Follow style conventions you see used in the source already (and read PEP8).
- Include tests that fail *without* your code, and pass *with* it. Only minor refactoring and documentation changes require no new tests. If you are adding functionality or fixing a bug, please also add a test for it!
- Update any documentation or examples impacted by your change.
- Styling conventions and code quality are checked with `invoke check`, tests are run using `invoke test`, and the docs can be built locally using `invoke build --docs`.

Following these hints also expedites the whole procedure, since it avoids unnecessary feedback cycles.

4.6 Software License

4.6.1 Full License Text

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.
```

(continues on next page)

(continued from previous page)

```
   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

CHAPTER

FIVE

REFERENCES

5.1 Tools

- Cookiecutter
- PyInvoke
- pytest
- tox
- Pylint
- twine
- bpython
- yolk3k

5.2 Packages

- Rituals

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX