Rose Tutorial

Release 2019.01.0

Metomi

Jan 25, 2019

CONTENTS

Rose is a toolkit for writing, editing and running application configurations.

Rose also contains other optional tools for:

- Version control.
- Suite discovery and management.
- Validating and transforming Rose configurations.
- Interfacing with Cylc.

CHAPTER

ONE

ROSE CONFIGURATIONS

Rose configurations are directories containing a Rose configuration file along with other optional assets which define behaviours such as:

- Execution.
- File installation.
- Environment variables.

Rose configurations may be used standalone or alternatively in combination with the Cylc (http://cylc.github.io/cylc/) workflow engine. There are two types of Rose configuration for use with Cylc (http://cylc.github.io/cylc/):

Rose application configuration A runnable Rose configuration which executes a defined command.

Rose suite configuration A Rose configuration designed to run Cylc suites. For instance it may be used to define Jinja2 variables for use in the `suite.rc` file.

1.1 Rose Configuration Format

Rose configurations are directories containing a Rose configuration file along with other optional files and directories.

All Rose configuration files use the same format which is based on the INI (https://en.wikipedia.org/wiki/INI_file) file format. *Like* the file format for Cylc suites:

- Comments start with a `#` character.
- Settings are written as `key=value` pairs.
- Sections are written inside square brackets i.e. `[section-name]`

However, there are also key differences, and *unlike* the file format for Cylc suites:

- Sections cannot be nested.
- Settings should not be indented.
- Comments must start on a new line (i.e. you cannot have inline comments).
- There should not be spaces around the = operator in a `key=value` pair.

For example:

```
# Comment.
setting=value

[section]
key=value
multi-line-setting=multi
                  =line
                  =value
```

Hint: In Rose configuration files settings do not normally require quotation.

Throughout this tutorial we will refer to settings in the following format:

- `file` - will refer to a Rose configuration *file*.
- `file|setting` - will refer to a *setting* in a Rose configuration file.
- `file[section]` - will refer to a *section* in a Rose configuration file.
- `file[section]setting` - will refer to a *setting in a section* in a Rose configuration file.

1.2 Why Use Rose Configurations?

With Rose configurations the inputs and environment required for a particular purpose can be encapsulated in a simple human-readable configuration.

Configuration settings can have metadata associated with them which may be used for multiple purposes including automatic checking and transforming.

Rose configurations can be edited either using a text editor or with the command-rose-config-edit GUI which makes use of metadata for display and on-the-fly validation purposes.

CHAPTER

TWO

ROSE APPLICATIONS

The Cylc `suite.rc` file allows us to define environment variables for use by tasks e.g:

```
[runtime]
    [[hello_world]]
        script = echo "Hello ${WORLD}!"
        [[[environment]]]
            WORLD = Earth
```

As a task grows in complexity it could require:

- More environment variables.
- Input files.
- Scripts and libraries.

A Rose application or "Rose app" is a runnable Rose configuration which executes a defined command.

Rose applications provide a convenient way to encapsulate all of this configuration, storing it all in one place to make it easier to handle and maintain.

2.1 Application Configuration

An application configuration is a directory containing a `rose-app.conf` file. Application configurations are also referred to as "applications" or "apps".

The command to execute when the application is run is defined using the `rose-app.conf[command]default` setting e.g:

```
[command]
default=echo "Hello ${WORLD}!"
```

Environment variables are specified inside the `rose-app.conf[env]` section e.g:

```
[env]
WORLD=Earth
```

Scripts and executables can be placed in a `bin/` directory. They will be automatically added to the `PATH` environment variable when the application is run, e.g.:

Listing 1: bin/hello

```
echo "Hello ${WORLD}!"
```

Listing 2: rose-app.conf

```
[command]
default=hello
```

Any static input files can be placed in the `file/` directory.

2.2 Running Rose Applications

An application can be run using the command-rose-app-run command:

```
$ rose app-run -q  # -q for quiet output
Hello Earth!
```

The Rose application will by default run in the current directory so it is a good idea to run it outside of the application directory to keep run files separate, using the `-C` option to provide the path to the application:

```
$ rose app-run -q -C path/to/application
Hello Earth!
```

Practical

In this practical we will convert the `forecast` task from the weather-forecasting suite into a Rose application.

Create a directory on your filesystem called `rose-tutorial`:

```
mkdir ~/rose-tutorial
cd ~/rose-tutorial
```

1. **Create a Rose application**

   Create a new directory called `application-tutorial`, this is to be our application directory:

   ```
   mkdir application-tutorial
   cd application-tutorial
   ```

2. **Move the required resources into the `application-tutorial` application.**

   The application requires three resources:

   - The `bin/forecast` script.
   - The `lib/python/util.py` Python library.
   - The `lib/template/map.html` HTML template.

   Rather than leaving these resources scattered throughout the suite directory we can encapsulate them into the application directory.

   Copy the `forecast` script and `util.py` library into the `bin/` directory where they will be automatically added to the `PATH` when the application is run:

   ```
   rose tutorial forecast-script bin
   ```

   Copy the HTML template into the `file/` directory by running:

   ```
   rose tutorial map-template file
   ```

3. **Create the `rose-app.conf` file.**

   The `rose-app.conf` file needs to define the command to run. Create a `rose-app.conf` file directly inside the application directory containing the following:

```
[command]
default=forecast $INTERVAL $N_FORECASTS
```

   The `INTERVAL` and `N_FORECASTS` environment variables need to be defined. To do this add a `rose-app.conf[env]` section to the file:

```
[env]
# The interval between forecasts.
INTERVAL=60
# The number of forecasts to run.
N_FORECASTS=5
```

4. **Copy the test data.**

   For now we will run the `forecast` application using some sample data so that we can run it outside of the weather forecasting suite.

   The test data was gathered in November 2017.

   Copy the test data files into the `file/` directory by running:

```
rose tutorial test-data file/test-data
```

5. **Move environment variables defined in the `suite.rc` file.**

   In the `[runtime][forecast][environment]` section of the `suite.rc` file in the weather-forecasting suite we set a few environment variables:

   - `WIND_FILE_TEMPLATE`
   - `WIND_CYCLES`
   - `RAINFALL_FILE`
   - `MAP_FILE`
   - `MAP_TEMPLATE`

   We will now move these into the application. This way, all of the configuration specific to the application live within it.

   Add the following lines to the `rose-app.conf[env]` section:

```
# The weighting to give to the wind file from each WIND_CYCLE
# (comma separated list, values should add up to 1).
WEIGHTING=1
# Comma separated list of cycle points to get wind data from.
WIND_CYCLES=0
# Path to the wind files. {cycle}, {xy} will get filled in by the
# forecast script.
WIND_FILE_TEMPLATE=test-data/wind_{cycle}_{xy}.csv
# Path to the rainfall file.
RAINFALL_FILE=test-data/rainfall.csv
# The path to create the HTML map in.
MAP_FILE=map.html
# The path to the HTML map template file.
MAP_TEMPLATE=map-template.html
```

   Note that the `WIND_FILE_TEMPLATE` and `RAINFALL_FILE` environment variables are pointing at files in the `test-data` directory.

To make this application work outside of the weather forecasting suite we will also need to provide the `DOMAIN` and `RESOLUTION` environment variables defined in the `[runtime][root][environment]` section of the `suite.rc` file as well as the `CYLC_TASK_CYCLE_POINT` environment variable provided by Cylc when it runs a task.

Add the following lines to the `rose-app.conf`:

```
# The date when the test data was gathered.
CYLC_TASK_CYCLE_POINT=20171101T0000Z
# The dimensions of each grid cell in degrees.
RESOLUTION=0.2
# The area to generate forecasts for (lng1, lat1, lng2, lat2).
DOMAIN=-12,48,5,61
```

6. **Run the application.**

   All of the scripts, libraries, files and environment variables required to make a forecast are now provided inside this application directory.

   We should now be able to run the application.

   command-rose-app-run will run an application in the current directory so it is a good idea to move somewhere else before calling the command. Create a directory and run the application in it:

```
mkdir run
cd run
rose app-run -C ../
```

   The application should run successfully, leaving behind some files. Try opening the `map.html` file in a web browser.

CHAPTER

THREE

ROSE METADATA

Metadata can be used to provide information about settings in Rose configurations.

It is used for:

- Documenting settings.
- Performing automatic checking (e.g. type checking).
- Formatting the command-rose-config-edit GUI.

Metadata can be used to ensure that configurations are valid before they are run and to assist those who edit the configurations.

3.1 The Metadata Format

Metadata is written in a `rose-meta.conf` file. This file can either be stored inside a Rose configuration in a `meta/` directory, or elsewhere outside of the configuration.

The `rose-meta.conf` file uses the standard *Rose configuration format* (page 3).

The metadata for a setting is written in a section named `[section=setting]` where `setting` is the name of the setting and `section` is the section to which the setting belongs (left blank if the setting does not belong to a section).

For example, take the following application configuration:

Listing 1: rose-app.conf

```
[command]
default=echo "Hello ${WORLD}."

[env]
WORLD=Earth
```

If we were to write metadata for the `WORLD` environment variable we would create a section called `[env=WORLD]`.

Listing 2: meta/rose-meta.conf

```
[env=WORLD]
description=The name of the world to say hello to.
values=Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
```

This example gives the `WORLD` variable a title and a list of allowed values.

3.2 Metadata Commands

The command-rose-metadata-check command can be used to check that metadata is valid:

```
$ rose metadata-check -C meta/
```

The configuration can be tested against the metadata using the `-V` option of the command-rose-macro command.

For example, if we were to change the value of `WORLD` to `Pluto`:

```
$ rose macro -V
Value Pluto not in allowed values ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
↪'Saturn', 'Uranus', 'Neptune']
```

3.3 Metadata Items

There are many metadata items, some of the most commonly-used ones being:

`title` Assign a title to a setting.

`description` Attach a short description to a setting.

`type` Specify the data type a setting expects, e.g. `type=integer`.

`length` Specify the length of comma-separated lists, e.g. `length=:` for a limitless list.

`range` Specify numerical bounds for the value of a setting, e.g. `range=1, 10` for a value between 1 and 10.

For a full list of metadata items, see `rose-meta.conf[SETTING]`.

Practical

In this practical we will write metadata for the `application-tutorial` app we wrote in the Rose application practical.

1. **Create a Rose application called** `metadata-tutorial`.

   Create a new copy of the `application-tutorial` application by running:

   ```
   rose tutorial metadata-tutorial ~/rose-tutorial/metadata-tutorial
   cd ~/rose-tutorial/metadata-tutorial
   ```

2. **View the application in** command-rose-config-edit.

   The command-rose-config-edit command opens a GUI which displays Rose configurations. Open the `metadata-tutorial` app:

   ```
   rose config-edit &
   ```

   Tip: Note command-rose-config-edit searches for any Rose configuration in the current directory. Use the `-C` option to specify another directory.

   In the panel on the left you will see the different sections of the `rose-app.conf` file.

   Click on *env*, where you will find all of the environment variables. Each setting will have a hash symbol (#) next to its name. These are the comments defined in the `rose-app.conf` file. Hover the mouse over the hash to reveal the comment.

   Keep the command-rose-config-edit window open as we will use it throughout the rest of this practical.

3. **Add descriptions.**

   Now we will start writing some metadata.

   Create a `meta/` directory containing a `rose-meta.conf` file:

```
mkdir meta
touch meta/rose-meta.conf
```

   In the `rose-app.conf` file there are comments associated with each setting. Take these comments out of the `rose-app.conf` file and add them as descriptions in the metadata. As an example, for the `INTERVAL` environment variable you would create a metadata entry that looks like this:

```
[env=INTERVAL]
description=The interval between forecasts.
```

   Longer settings can be split over multiple lines like so:

```
[env=INTERVAL]
description=The interval
           =between forecasts.
```

   Once you have finished save your work and validate the metadata using command-rose-metadata-check:

```
rose metadata-check -C meta/
```

   There should not be any errors so this check will silently pass.

   Next reload the metadata in the command-rose-config-edit window using the *Metadata → Refresh Metadata* menu item. The descriptions should now display under each environment variable.

Tip: If you don't see the description for a setting it is possible that you misspelt the name of the setting in the section heading.

---

4. **Indicate list settings and their length.**

   The `DOMAIN` and `WEIGHTING` settings both accept comma-separated lists of values. We can represent this in Rose metadata using the `rose-meta.conf[SETTING]length` setting.

   To represent the `DOMAIN` setting as a list of four elements, add the following to the `[env=DOMAIN]` section:

```
length=4
```

   The `WEIGHTING` and `WIND_CYCLES` settings are different as we don't know how many items they will contain. For flexible lists we use a colon, so add the following line to the `[env=WEIGHTING]` and `[env=WIND_CYCLES]` sections:

```
length=:
```

   Validate the metadata:

```
rose metadata-check -C meta/
```

   Refresh the metadata in the command-rose-config-edit window by selecting *Metadata → Refresh Metadata*. The three settings we have edited should now appear as lists.

5. **Specify data types.**

   Next we will add type information to the metadata.

   The `INTERVAL` setting accepts an integer value. Add the following line to the `[env=INTERVAL]` section to enforce this:

```
type=integer
```

Validate the metadata and refresh the command-rose-config-edit window. The `INTERVAL` setting should now appear as an integer rather than a text field.

In the command-rose-config-edit window, try changing the value of `INTERVAL` to a string. It shouldn't let you do so.

Add similar `type` entries for the following settings:

`integer` settings	`real` (float) settings
INTERVAL	WEIGHTING
N_FORECASTS	RESOLUTION

Validate the metadata to check for errors.

In the command-rose-config-edit window try changing the value of `RESOLUTION` to a string. It should be marked as an error.

6. **Define sets of allowed values.**

   We will now add a new input to our application called `SPLINE_LEVEL`. This is a science setting used to determine the interpolation method used on the rainfall data. It accepts the following values:

   - `0` - for nearest member interpolation.
   - `1` - for linear interpolation.

   Add this setting to the `rose-app.conf` file:

```
[env]
SPLINE_LEVEL=0
```

   We can ensure that users stick to allowed values using the `values` metadata item. Add the following to the `rose-meta.conf` file:

```
[env=SPLINE_LEVEL]
values=0,1
```

   Validate the metadata.

   As we have made a change to the configuration (by editing the `rose-app.conf` file) we will need to close and reload the command-rose-config-edit GUI. The setting should appear as a button with only the options `0` and `1`.

   Unfortunately `0` and `1` are not particularly descriptive, so it might not be obvious that they mean "nearest" and "linear" respectively. The `rose-meta.conf[SETTING]value-titles` metadata item can be used to add titles to such settings to make the values clearer.

   Add the following lines to the `[env=SPLINE_LEVEL]` section in the `rose-meta.conf` file:

```
value-titles=Nearest,Linear
```

   Validate the metadata and refresh the command-rose-config-edit window. The `SPLINE_LEVEL` options should now have titles which better convey the meaning of the options.

   Tip: The `rose-meta.conf[SETTING]value-hints` metadata option can be used to provide a longer description of each option.

   ---

7. **Validate with** `rose macro`.

On the command line command-rose-macro can be used to check that the configuration is compliant with the metadata. Try editing the `rose-app.conf` file to introduce errors then validating the configuration by running:

```
rose macro -V
```

CHAPTER

FOUR

ROSE SUITE CONFIGURATIONS

Rose application configurations can be used to encapsulate the environment and resources required by a Cylc task.

Similarly Rose suite configurations can be used to do the same for a Cylc suite.

4.1 Configuration Format

A Rose suite configuration is a Cylc suite directory containing a `rose-suite.conf` file.

The `rose-suite.conf` file is written in the same *format* (page 3) as the `rose-app.conf` file. Its main configuration sections are:

`rose-suite.conf[env]` Environment variables for use by the whole suite.

`rose-suite.conf[jinja2:suite.rc]` Jinja2 (http://jinja.pocoo.org/) variables for use in the `suite.rc` file.

`rose-suite.conf[empy:suite.rc]` EmPy (http://www.alcyone.com/software/empy/) variables for use in the `suite.rc` file.

`rose-suite.conf[file:NAME]` Files and resources to be installed in the run directory when the suite is run.

In the following example the environment variable `GREETING` and the Jinja2 variable `WORLD` are both set in the `rose-suite.conf` file. These variables can then be used in the `suite.rc` file:

Listing 1: rose-suite.conf

```
[env]
GREETING=Hello

[jinja2:suite.rc]
WORLD=Earth
```

Listing 2: suite.rc

```
[scheduling]
    [[dependencies]]
        graph = hello_{{WORLD}}

[runtime]
    [[hello_{{WORLD}}]]
        script = echo "$GREETING {{WORLD}}"
```

4.2 Suite Directory Vs Run Directory

suite directory The directory in which the suite is written. The `suite.rc` and `rose-suite.conf` files live here.

run directory The directory in which the suite runs. The `work`, `share` and `log` directories live here.

Throughout the Cylc Tutorial we wrote suites in the `cylc-run` directory. As Cylc runs suites in the `cylc-run` directory the suite directory is also the run directory i.e. the suite runs in the same directory in which it is written.

With Rose we develop suites in a separate directory to the one in which they run meaning that the suite directory is different from the run directory. This helps keep the suite separate from its output and means that you can safely work on a suite and its resources whilst it is running.

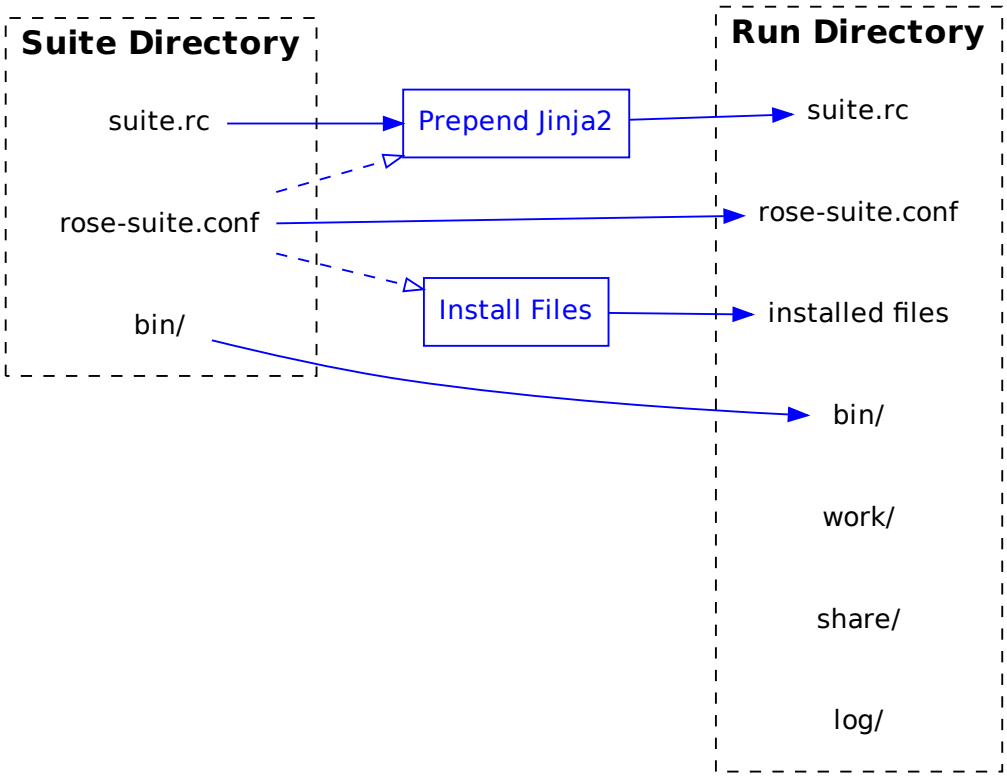
Note: Using Cylc it is possible to separate the suite directory and run directory using the `cylc register` command. Note though that suite resources, e.g. scripts in the `bin/` directory, will remain in the suite directory so cannot safely be edited whilst the suite is running.

4.3 Running Rose Suite Configurations

Rose *Application Configurations* (page 5) are run using command-rose-app-run, Rose Suite Configurations are run using command-rose-suite-run.

When a suite configuration is run:

1. The suite directory is copied into the `cylc-run` directory where it becomes the run directory.
2. Any files defined in the `rose-suite.conf` file are installed.
3. Jinja2 variables defined in the `rose-suite.conf` file are added to the top of the `suite.rc` file.
4. The Cylc suite is validated.
5. The Cylc suite is run.
6. The Cylc GUI is launched.



Like command-rose-app-run, command-rose-suite-run will look for a configuration to run in the current directory. The command can be run from other locations using the `-C` argument:

```
rose suite-run -C /path/to/suite/configuration/
```

The `--local-install-only` command line option will cause the suite to be installed (though only on your local machine, not on any job hosts) and validated but not run (i.e. *steps 1-4* (page 16)).

4.4 Start, Stop, Restart

Under Rose, suites will run using the name of the suite directory. For instance if you run command-rose-suite-run on a suite in the directory `~/foo/bar` then it will run with the name `bar`.

The name can be overridden using the `--name` option i.e:

```
rose suite-run --name <SUITE_NAME>
```

Starting Suites Suites must be run using the command-rose-suite-run command which in turn calls the `cylc run` command.

Stopping Suites Suites can be stopped using the `cylc stop <SUITE_NAME>` command, as for regular Cylc suites.

Restarting Suites There are two options for restarting:

- To pick up where the suite left off use command-rose-suite-restart. No changes will be made to the run directory. *This is usually the recommended option.*

- To restart in a way that picks up changes made in the suite directory, use the `--restart` option to the command-rose-suite-run command.

See the Cheat Sheet for more information.

Note: command-rose-suite-run installs suites to the run directory incrementally so if you change a file and restart the suite using `rose suite-run --restart` only the changed file will be re-installed. This process is strictly constructive i.e. any files deleted in the suite directory will *not* be removed from the run directory. To force command-rose-suite-run to perform a complete rebuild, use the `--new` option.

Practical

In this tutorial we will create a Rose Suite Configuration for the weather-forecasting suite.

1. **Create A New Suite.**

   Create a copy of the weather-forecasting suite by running:

   ```
   rose tutorial rose-suite-tutorial ~/rose-tutorial/rose-suite-tutorial
   cd ~/rose-tutorial/rose-suite-tutorial
   ```

   Set the initial and final cycle points as in previous tutorials.

2. **Create A Rose Suite Configuration.**

   Create a blank `rose-suite.conf` file:

   ```
   touch rose-suite.conf
   ```

   You now have a Rose suite configuration. A `rose-suite.conf` file does not need to have anything in it but it is required to run command-rose-suite-run.

   There are three things defined in the `suite.rc` file which it might be useful to be able to configure:

   `station` The list of weather stations to gather observations from.

   `RESOLUTION` The spatial resolution of the forecast model.

   `DOMAIN` The geographical limits of the model.

   Define these settings in the `rose-suite.conf` file by adding the following lines:

   ```
   [jinja2:suite.rc]
   station="camborne", "heathrow", "shetland", "belmullet"

   [env]
   RESOLUTION=0.2
   DOMAIN=-12,48,5,61
   ```

   Note that Jinja2 (http://jinja.pocoo.org/) strings must be quoted.

3. **Write Suite Metadata.**

   Create a `meta/rose-meta.conf` file and write some metadata for the settings defined in the `rose-suite.conf` file.

   - `station` is a list of unlimited length.
   - `RESOLUTION` is a "real" number.
   - `DOMAIN` is a list of four integers.

Tip: For the `RESOLUTION` and `DOMAIN` settings you can copy the metadata you wrote in the *Metadata Tutorial* (page 9).

Solution

```
[jinja2:suite.rc=station]
length=:

[env=RESOLUTION]
type=real

[env=DOMAIN]
length=4
type=integer
```

Validate the metadata:

```
rose metadata-check -C meta/
```

Open the command-rose-config-edit GUI. You should see *suite conf* in the panel on the left-hand side of the window. This will contain the environment and Jinja2 variables we have just defined.

4. **Use Suite Variables In The `suite.rc` File.**

   Next we need to make use of these settings in the `suite.rc` file.

   We can delete the `RESOLUTION` and `DOMAIN` settings in the `[runtime][root][environment]` section which would otherwise override the variables we have just defined in the `rose-suite.conf` file, like so:

```
 [runtime]
     [[root]]
         # These environment variables will be available to all tasks.
         [[[environment]]]
             # Add the `python` directory to the PYTHONPATH.
             PYTHONPATH="$CYLC_SUITE_RUN_DIR/lib/python:$PYTHONPATH"
-            # The dimensions of each grid cell in degrees.
-            RESOLUTION = 0.2
-            # The area to generate forecasts for (lng1, lat1, lng2, lat2).
-            DOMAIN = -12,48,5,61  # Do not change!
```

   We can write out the list of stations, using the Jinja2 (http://jinja.pocoo.org/) `join` filter to write the commas between the list items:

```
 [cylc]
     UTC mode = True
     [[parameters]]
         # A list of the weather stations we will be fetching observations␣
↪from.
-        station = camborne, heathrow, shetland, belmullet
+        station = {{ station | join(", ") }}
         # A list of the sites we will be generating forecasts for.
         site = exeter
```

5. **Install The Suite.**

   Running command-rose-suite-run will cause the suite to be installed, validated and run.

   Use the `--local-install-only` command-line option to install the suite on your local machine and validate it:

```
rose suite-run --local-install-only
```

Inspect the installed suite, which you will find in the run directory, i.e:

```
~/cylc-run/rose-suite-tutorial
```

You should find all the files contained in the suite directory as well as the run directory folders `log`, `work` and `share`.

4.5 Rose Applications In Rose Suite Configurations

In Cylc suites, Rose applications are placed in an `app/` directory which is copied across to the run directory with the rest of the suite by command-rose-suite-run when the suite configuration is run.

When we run Rose applications from within Cylc suites we use the command-rose-task-run command rather than the command-rose-app-run command.

When run, command-rose-task-run searches for an application with the same name as the Cylc task in the `app/` directory.

The command-rose-task-run command also interfaces with Cylc to provide a few useful environment variables (see the command-line reference for details). The application will run in the work directory, just like for a regular Cylc task.

In this example the `hello` task will run the application located in `app/hello/`:

Listing 3: suite.rc

```
[runtime]
    [[hello]]
        script = rose task-run
```

Listing 4: app/hello/rose-app.conf

```
[command]
default=echo "Hello World!"
```

The name of the application to run can be overridden using the `--app-key` command-line option or the `ROSE_TASK_APP` environment variable. For example the `greetings` task will run the `hello` app in the task defined below.

Listing 5: suite.rc

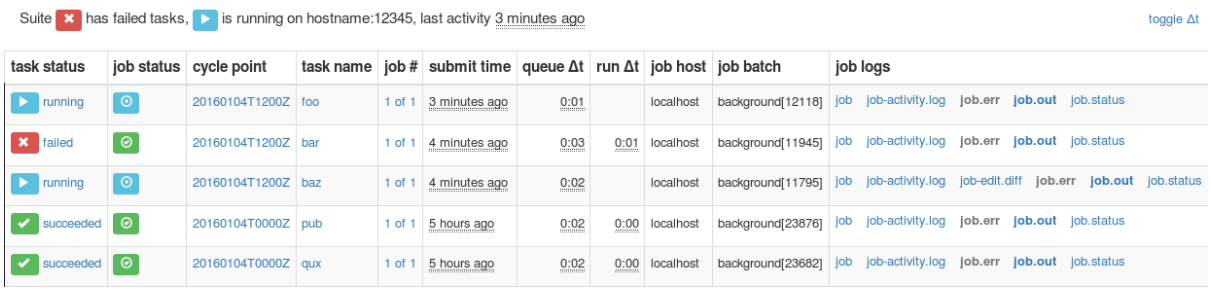
```
[runtime]
    [[greetings]]
        script = rose task-run --app-key hello
```

4.6 Rose Bush

Rose provides a utility for viewing the status and logs of Cylc suites called Rose Bush. Rose Bush displays suite information in web pages.

If a Rose Bush server is provided at your site, you can open the Rose Bush page for a suite by running the command-rose-suite-log command in the suite directory.

Otherwise an add-hoc web server can be set up using the command-rose-bush `start` command argument.


Suite has failed tasks, is running on hostname:12345, last activity 3 minutes ago

toggle Δt

task status	job status	cycle point	task name	job #	submit time	queue Δt	run Δt	job host	job batch	job logs
running		20160104T1200Z	foo	1 of 1	3 minutes ago	0:01		localhost	background[12118]	job job-activity.log job.err job.out job.status
failed		20160104T1200Z	bar	1 of 1	4 minutes ago	0:03	0:01	localhost	background[11945]	job job-activity.log job.err job.out job.status
running		20160104T1200Z	baz	1 of 1	4 minutes ago	0:02		localhost	background[11795]	job job-activity.log job-edit.diff job.err job.out job.status
succeeded		20160104T0000Z	pub	1 of 1	5 hours ago	0:02	0:00	localhost	background[23876]	job job-activity.log job.err job.out job.status
succeeded		20160104T0000Z	qux	1 of 1	5 hours ago	0:02	0:00	localhost	background[23682]	job job-activity.log job.err job.out job.status

Fig. 1: Screenshot of a Rose Bush web page.

Practical

In this practical we will take the `forecast` Rose application that we developed in the Metadata Tutorial and integrate it into the weather-forecasting suite.

Move into the suite directory from the previous practical:

```
cd ~/rose-tutorial/rose-suite-tutorial
```

You will find a copy of the `forecast` application located in `app/forecast`.

1. **Create A Test Configuration For The `forecast` Application.**

   We have configured the `forecast` application to use test data. We will now adjust this configuration to make it work with real data generated by the Cylc suite. It is useful to keep the ability to run the application using test data, so we won't delete this configuration. Instead we will move it into an Optional Configuration so that we can run the application in "test mode" or "live mode".

   Optional configurations are covered in more detail in the *Optional Configurations Tutorial* (page 49). For now all we need to know is that they enable us to store alternative configurations.

   Create an optional configuration called `test` inside the `forecast` application:

   ```
   mkdir app/forecast/opt
   touch app/forecast/opt/rose-app-test.conf
   ```

   This optional configuration is a regular Rose configuration file. Its settings will override those in the `rose-app.conf` file if requested.

   Tip: Take care not to confuse the `rose-app.conf` and `rose-app-test.conf` files used within this practical.

   Move the following environment variables from the `app/forecast/rose-app.conf` file into an `[env]` section in the `app/forecast/opt/rose-app-test.conf` file:

   - `WEIGHTING`
   - `WIND_CYCLES`
   - `WIND_FILE_TEMPLATE`
   - `RAINFALL_FILE`
   - `MAP_FILE`
   - `CYLC_TASK_CYCLE_POINT`
   - `RESOLUTION`

- DOMAIN

Solution

The `rose-app-test.conf` file should look like this:

```
[env]
WEIGHTING=1
WIND_CYCLES=0
WIND_FILE_TEMPLATE=test-data/wind_{cycle}_{xy}.csv
RAINFALL_FILE=test-data/rainfall.csv
MAP_FILE=map.html
CYLC_TASK_CYCLE_POINT=20171101T0000Z
RESOLUTION=0.2
DOMAIN=-12,48,5,61
```

Run the application in "test mode" by providing the option `--opt-conf-key=test` to the command-rose-app-run command:

```
mkdir app/forecast/run
cd app/forecast/run
rose app-run --opt-conf-key=test -C ../
cd ../../../
```

You should see the stdout output of the Rose application. If there are any errors they will be marked with the `[FAIL]` prefix.

2. **Integrate The `forecast` Application Into The Suite.**

   We can now configure the `forecast` application to work with real data.

   We have moved the map template file (`map-template.html`) into the `forecast` application so we can delete the `MAP_TEMPLATE` environment variable from the `[runtime]forecast` section of the `suite.rc` file.

   Copy the remaining environment variables defined in the `forecast` task within the `suite.rc` file into the `rose-app.conf` file of the `forecast` application, replacing any values already specified if necessary. Remove the lines from the `suite.rc` file when you are done.

   Remember, in Rose configuration files:

   - Spaces are not used around the equals (=) operator.
   - Ensure the environment variables are not quoted.

   The `[env]` section of your `rose-app.conf` file should now look like this:

```
[env]
INTERVAL=60
N_FORECASTS=5
WEIGHTING=1
MAP_TEMPLATE=map-template.html
SPLINE_LEVEL=0
WIND_FILE_TEMPLATE=$CYLC_SUITE_WORK_DIR/{cycle}/consolidate_observations/wind_
↪{xy}.csv
WIND_CYCLES=0, -3, -6
RAINFALL_FILE=$CYLC_SUITE_WORK_DIR/$CYLC_TASK_CYCLE_POINT/get_rainfall/
↪rainfall.csv
MAP_FILE=${CYLC_TASK_LOG_ROOT}-map.html
```

   Finally we need to change the `forecast` task to run command-rose-task-run. The `[runtime]forecast` section of the `suite.rc` file should now look like this:

```
[[forecast]]
    script = rose task-run
```

3. **Make Changes To The Configuration.**

   Open the command-rose-config-edit GUI and navigate to the *suite conf* > *env* panel.

   Change the `RESOLUTION` variable to `0.1`

   Navigate to the *forecast* > *env* panel.

   Edit the `WEIGHTING` variable so that it is equal to the following list of values:

```
0.7, 0.2, 0.1
```

   Tip: Click the "Add array element" button (+) to extend the number of elements assigned to `WEIGHTING`.

   Finally, save these settings via *File* > *Save* in the menu.

4. **Run The Suite.**

   Install, validate and run the suite:

```
rose suite-run
```

   The `cylc gui` should open and the suite should run and complete.

5. **View Output In Rose Bush.**

   Open the Rose Bush page in a browser by running the following command from within the suite directory:

```
rose suite-log
```

   On this page you will see the tasks run by the suite, ordered from most to least recent. Near the top you should see an entry for the `forecast` task. On the right-hand side of the screen click *job-map.html*.

   As this file has a `.html` extension Rose Bush will render it. The raw text would be displayed otherwise.

CHAPTER

FIVE

ROSIE

Rosie is a tool for managing Rose suite configurations which is included in Rose. The purpose of Rosie is to facilitate suite development, management and collaboration. Rosie:

- Adds version control to Rose suite configurations.
- Updates a database to keep track of Rose suite configurations.

Warning: This tutorial does not require specific FCM knowledge but basic version control awareness is important. For more information on FCM version control see the FCM User Guide (http://metomi.github.io/fcm/doc/user_guide/).

5.1 Rosie Suites

A Rosie suite is a Rose suite configuration which is managed by the Rosie system.

Rosie suites can be created by the command:

command-rosie-create Create a new suite or copy an existing one.

By default Rosie creates the working copy (http://svnbook.red-bean.com/en/1.7/svn.basic.in-action.html#svn.basic.in-action.wc) (local copy) of new suites in the `~/roses` directory though Rosie working copies can be created elsewhere.

Working copy installed in `~/roses`.

5.2 Version Control

In Rosie suites the suite directory is added to *version control* (page 25) using FCM (https://metomi.github.io/fcm/doc/).

FCM is a subversion (SVN) wrapper which provides a standard working practice for SVN projects. FCM implements all of the SVN commands as well as additional functionality. See the FCM User Guide (http://metomi.github.io/fcm/doc/user_guide/) for more information.

5.3 Suite Naming

Each Rosie suite is assigned a unique name made up of a *prefix* followed by a hyphen and then an *identifier* made up of two characters and three numbers, e.g:

u - aa001

Prefix Unique Identifier

The prefix denotes the repository in which the suite is located. Prefixes are site specific and are configured by the `rose.conf[rosie-id]prefix-location.PREFIX` setting.

Within the Rose user community the `u` prefix is typically configured to point at the SRS (https://code.metoffice.gov.uk/) repository.

5.4 The `rose-suite.info` File

All Rosie suites require a `rose-suite.info` file. This file provides information about the suite for use in the suite management and version control systems. The `rose-suite.info` file uses the *Rose Configuration Format* (page 3). The main settings are:

`title` A short title for the suite.

`owner` The user who has control over the suite (i.e. their username).

`project` The project to which this suite belongs (can be an arbitrary name).

`access-list` An optional list of users who have permission to commit to the trunk of the suite.

5.5 Managing Suites

Rosie provides commands for managing suites, including:

command-rosie-checkout Creates a local copy of a suite.

command-rosie-ls Lists all locally checked-out suites.

command-rosie-lookup Searches the suite database (using information from suite's `rose-suite.info` files).

Rosie also provides a GUI called command-rosie-go which incorporates the functionality of the above commands.

Practical

In this practical we will add the weather-forecasting suite from the previous practical to a rosie repository, make some changes, and commit them to the repository.

Note: For brevity this practical uses the abbreviated version of SVN commands, e.g. `svn st` is the abbreviated form of `svn status`. FCM supports both the full and abbreviated names.

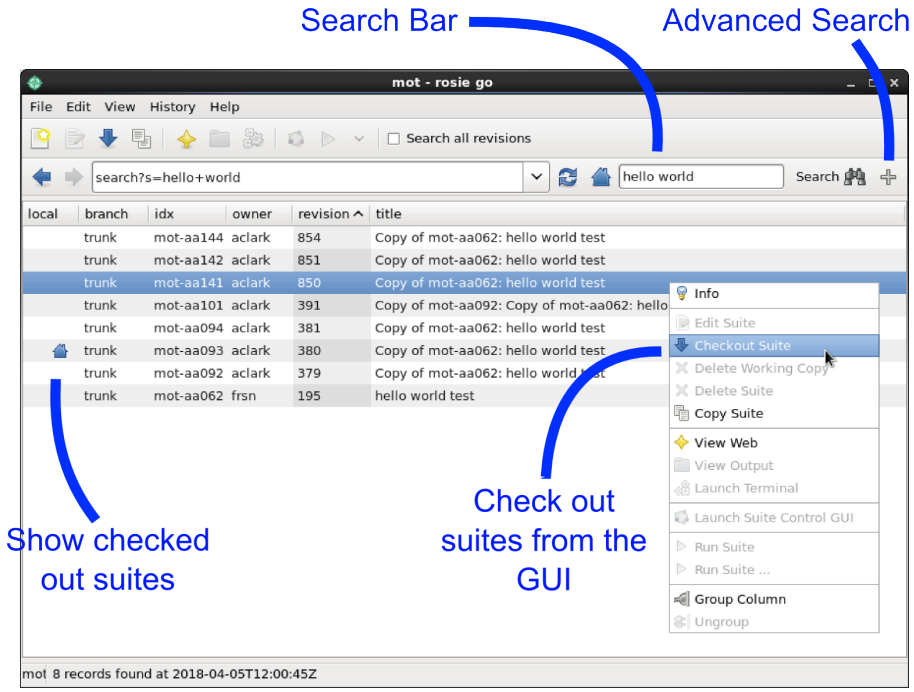


Fig. 1: Screenshot of the rosie go GUI.

1. **Create A New Rosie Suite.**

   First, create a blank Rosie suite in an appropriate repository. You will probably want to use a "testing" repository if one is available to you.

   You can specify the repository to use with the `--prefix` command-line option. For instance to use the (internal) Met Office Testing Repository supply the command line argument `--prefix=mot`.

```
rosie create --prefix=<prefix>
```

   You will then be presented with a `rose-suite.info` file open in a text editor. For the `title` field type "Dummy Weather Forecasting Suite" and for the `project` enter "tutorial". Save the file and close the editor.

   Tip: If the text editor does not appear you may have to press enter on the keyboard.

   Rosie will create the new suite in the `~/roses` directory and the exact location will appear in the command output. Move into the suite directory:

```
cd ~/roses/<name>
```

2. **Add Files To The Suite.**

   Add the files from the Weather Forecasting Suite by running:

```
rose tutorial rose-weather-forecasting-suite .
```

   We now need to add these files to version control. First check the SVN status by running:

```
fcm st
```

   You should see a list of files with the `?` symbol next to them, as well as `rose-suite.conf` with an `M` symbol beside it. `?` means the files marked are untracked (not version controlled), whereas `M` indicates files which have been modified. Add all untracked files to version control by running:

```
fcm add --check .
```

Answer yes ("y") where prompted. Now check the status again:

```
fcm st
```

You should see a list of files with the `A` character, meaning "added", next to them. Finally commit the changes by running:

```
fcm ci
```

A text editor will open. Add a message for your commit, save the file and close the editor. You will then be prompted as to whether you want to make the commit. Answer yes.

You have now added the Weather Forecasting Suite to version control. Open the Trac browser to see your suite:

```
fcm browse
```

A web browser window will open, showing the Trac page for your Rosie suite.

3. **Find The Suite In Rosie Go.**

   Open the command-rosie-go GUI:

   ```
   rosie go &
   ```

   Open the advanced search options by clicking the add (+) button in the top right-hand corner of the window.

   Search for suites which you have authored by selecting *author* and filling in your username in the right-hand box:

   Press *Search*. You should see your suite appear with a home icon next to it, meaning that you have a local copy checked out.

   Right-click on the suite and then click *Info*. You should see the information defined in the `rose-suite.info` file.

   Help

   If your suite does not show up, select the menu item *Edit* → *Data Source* and ensure the repository you committed to is checked.

4. **Checkout The Suite.**

   Now that the suite is in the Rosie repository a working copy can be checked out on any machine with access to the repository by executing:

   ```
   rosie checkout <name>
   ```

   Test this by deleting the working copy then checking out a new one:

   ```
   cd ~/roses
   rm -rf <name>
   rosie checkout <name>
   ```

Practical Extension

1. **Make Changes In A Branch.**

   Next we will make a change to the suite. Rather than making this change in the "trunk" (referred to as "master" in git terminology) we will work in a new "branch".

   Create a new branch called "configuration-change" by running:

   ```
   fcm bc configuration-change
   ```

   Provide a brief commit message of your choosing when prompted and enter yes ("y").

   You can list all branches by running:

   ```
   fcm bls
   ```

   Switch to your new branch:

   ```
   fcm sw configuration-change
   ```

   Next, either using the command-rose-config-edit GUI or a text editor, change the `RESOLUTION` setting in the `rose-suite.conf` file to `0.1`.

   Check the status of the project:

   ```
   fcm st
   ```

   You should see the `rose-suite.conf` file with a `M`, meaning modified, next to it. Commit the change by running:

   ```
   fcm ci
   ```

   Again you will need to provide a commit message and answer yes to the prompt.

2. **Merge The Branch.**

   Switch back to the trunk then merge your change branch into the trunk:

   ```
   fcm sw trunk
   fcm merge configuration-change
   ```

   Check the status (you should see the `M` symbol next to the `rose-suite.conf` file) then commit the merge:

   ```
   fcm st
   fcm ci
   ```

CHAPTER

SIX

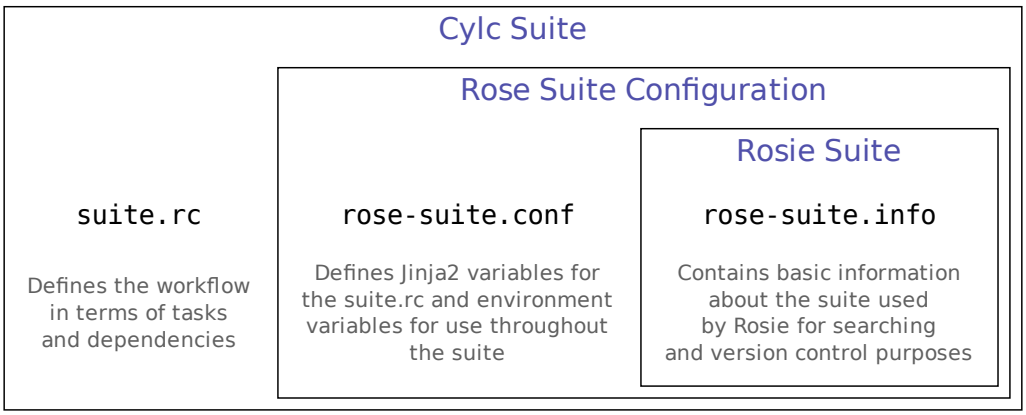
SUMMARY

6.1 Suite Structure

So far we have covered:

- Cylc suites.
- Rose suite configurations.
- Rosie suites.

The relationship between them is as follows:



Cylc suites can have Rose applications. These are stored in an `app` directory and are configured using a `rose-app.conf` file.

6.2 Suite Commands

We have learned the following Cylc commands:

`cylc graph` Draws the suite's graph.

`cylc get-config` Processes the `suite.rc` file and prints it back out.

`cylc validate` Validates the Cylc `suite.rc` file to check for any obvious errors.

`cylc run` Runs a suite.

`cylc stop` Stops a suite, in a way that:

`--kill` Kills all running/submitted tasks.

`--now --now` Leaves all running/submitted tasks running.

`cylc restart` Starts a suite, picking up where it left off from the previous run.

We have learned the following Rose commands:

command-rose-app-run Runs a Rose application.

command-rose-task-run Runs a Rose application from within a Cylc suite.

command-rose-suite-run Runs a Rose suite.

command-rose-suite-restart Runs a Rose suite, picking up where it left off from the previous run.

The Cylc commands do not know about the `rose-suite.conf` file so for Rose suite configurations you will have to install the suite before using commands such as `cylc graph`, e.g:

```
# install the suite on the local host only - don't run it.
rose suite-run --local-install-only

# run cylc graph using the installed version of the suite.
cylc graph <name>
```

6.3 Rose Utilities

Rose contains some utilities to make life easier:

command-rose-date A utility for parsing, manipulating and formatting date-times which is useful for working with the Cylc cycle point:

```
$ rose date 2000 --offset '+P1Y1M1D'
2001-02-02T0000Z

$ rose date $CYLC_TASK_CYCLE_POINT --format 'The month is %B.'
The month is April.
```

See the *date-time tutorial* (page 36) for more information.

command-rose-host-select A utility for selecting a host from a group with the ability to rank choices based on server load or free memory.

Groups are configured using the `rose.conf[rose-host-select]group{NAME}` setting. For example to define a cluster called "mycluster" containing the hosts "computer1", "computer2" and "computer3", you would write:

```
[rose-host-select]
group{mycluster}=computer1 computer2 computer3
```

Hosts can then be selected from the cluster on the command line:

```
$ rose host-select mycluster
computer2
```

The command-rose-host-select command can by used within Cylc suites to determine which host a task runs on:

```
[runtime]
    [[foo]]
        script = echo "Hello $(hostname)!"
        [[[remote]]]
            host = rose host-select mycluster
```

6.4 Rose Built-In Applications

Along with Rose utilities there are also Rose built-in applications.

`fcm_make` A template for running the `fcm make` command.

`rose_ana` Runs the rose-ana analysis engine.

`rose_arch` Provides a generic solution to configure site-specific archiving of suite files.

`rose_bunch` For the running of multiple command variants in parallel under a single job.

`rose_prune` A framework for housekeeping a cycling suite.

6.5 Next Steps

Further Topics **(page 35)** Tutorials going over some of the more specific aspects of Rose not covered in the main tutorial.

Cheat Sheet A quick breakdown of the commands for running and interacting with suites using Cylc and Rose.

Command Reference Contains the command-line documentation (also obtainable by calling `rose --help`).

Rose Configuration The possible settings which can be used in the different Rose configuration files.

Cylc Suite Design Guide **(http://cylc.github.io/cylc/doc/built-sphinx/suite-design-guide/suite-design-guide-master.html)** Contains recommended best practice for the style and structure of Cylc suites.

CHAPTER

SEVEN

FURTHER TOPICS

This section goes into detail in additional Rose topics.

7.1 Command Keys

This tutorial walks you through using command keys.

Command keys allow you to specify and run different commands for a Rose app.

They work just like the default command for an app but have to be specified explicitly as an option of command-rose-task-run.

7.1.1 Example

Create a new Rose suite configuration called `command-keys`:

```
mkdir -p ~/rose-tutorial/command-keys
cd ~/rose-tutorial/command-keys
```

Create a blank `rose-suite.conf` and a `suite.rc` file that looks like this:

```
[cylc]
    UTC mode = True # Ignore DST
[scheduling]
    [[dependencies]]
        graph = gather_ingredients => breadmaker

[runtime]
    [[gather_ingredients]]
        script = sleep 10; echo 'Done'
    [[breadmaker]]
        script = rose task-run
```

In your suite directory create an `app` directory.

In the `app` directory create a new directory called `breadmaker`.

In the `breadmaker` directory create a `rose-app.conf` file that looks like this:

```
[command]
default=sleep 10; echo 'fresh bread'
```

This sets up a simple suite that contains the following:

- A `breadmaker` app
- A `gather_ingredients` task
- A `breadmaker` task that runs the `breadmaker` app

Save your changes then run the suite using command-rose-suite-run.

Once it has finished use command-rose-suite-log to view the suite log. In the page that appears, click the "out" link for the breadmaker task. In the page you are taken to you should see a line saying "fresh bread".

7.1.2 Adding Alternative Commands

Open the `rose-app.conf` file and edit to look like this:

```
[command]
default=sleep 10; echo 'fresh bread'
make_dough=sleep 8; echo 'dough for later'
timed_bread=sleep 15; echo 'fresh bread when you want it'
```

Save your changes and open up your `suite.rc` file. Alter the `[[breadmaker]]` task to look like this:

```
[[breadmaker]]
    script=rose task-run --command-key=make_dough
```

Save your changes and run the suite. If you inspect the output from the breadmaker task you should see the line "dough for later".

Edit the script for the `[[breadmaker]]` task to change the command key to `timed_bread`. Run the suite and confirm the timed_bread command has been run.

7.1.3 Summary

You have successfully made use of command keys to run alternate commands in an app.

Possible uses of command keys might be:

- Running an app in different modes of verbosity
- Running an app in different configurations
- Specifying different options to an app
- During suite development to aid in debugging an app

7.2 Date and Time Manipulation

Datetime cycling suites inevitably involve performing some form of datetime arithmetic. In the weather forecasting suite we wrote in the Cylc tutorial this arithmetic was done using the `cylc cyclepoint` command. For example we calculated the cycle point three hours before the present cycle using:

```
cylc cyclepoint --offset-hours=-3
```

Rose provides the command-rose-date command which provides functionality beyond `cylc cyclepoint` as well as the `ROSE_DATAC` environment variable which provides an easy way to get the path of the `share/cycle` directory.

7.2.1 The `rose date` Command

The command-rose-date command provides functionality for:

- Parsing and formatting datetimes e.g:

```
$ rose date '12-31-2000' --parse-format='%m-%d-%Y'
12-31-2000
$ rose date '12-31-2000' --parse-format='%m-%d-%Y' --format='DD-MM-CCYY'
31-12-2000
```

- Adding offsets to datetimes e.g:

```
$ rose date '2000-01-01T0000Z' --offset '+P1M'
2000-02-01T0000Z
```

- Calculating the duration between two datetimes e.g:

```
$ rose date '2000' '2001'  # Note - 2000 was a leap year!
P366D
```

See the command-rose-date command reference for more information.

7.2.2 Using `rose date` In A Suite

In datetime cycling suites command-rose-date can work with the cyclepoint using the `CYLC_TASK_CYCLE_POINT` environment variable:

```
[runtime]
    [[hello_america]]
        script = rose date $CYLC_TASK_CYCLE_POINT --format='MM-DD-CCYY'
```

Alternatively, if you are providing the standard Rose task environment using command-rose-task-env then command-rose-date can use the `-c` option to pick up the cycle point:

```
[runtime]
    [[hello_america]]
        env-script = eval $(rose task-env)
        script = rose date -c --format='MM-DD-CCYY'
```

7.2.3 The `ROSE_DATAC` Environment Variable

There are two locations where task output is likely to be located:

The work directory Each task is executed within its work directory which is located in:

```
<run directory>/work/<cycle>/<task-name>
```

The path to a task's work directory can be obtained from the `CYLC_TASK_WORK_DIR` environment variable.

The share directory The share directory serves the purpose of providing a storage place for any files which need to be shared between different tasks.

Within the share directory data is typically stored within cycle subdirectories i.e:

```
<run directory>/share/<cycle>
```

These are called the `share/cycle` directories.

The path to the root of the share directory is provided by the `CYLC_SUITE_SHARE_DIR` environment variable so the path to the cycle subdirectory would be:

```
"$CYLC_SUITE_SHARE_DIR/$CYLC_SUITE_CYCLE_POINT"
```

The command-rose-task-env command provides the environment variable `ROSE_DATAC` which is a more convenient way to obtain the path of the `share/cycle` directory.

To get the path to a previous (or a future) `share/cycle` directory we can provide an offset to command-rose-task-env e.g:

```
rose task-env --cycle-offset=PT1H
```

The path is then made available as the `ROSE_DATACPT1H` environment variable.

7.3 Fail-If, Warn-If

Basic validation can be achieved using metadata settings such as `type` and `range`. The `fail-if` and `warn-if` metadata settings are scriptable enabling more advanced validation. They evaluate logical expressions, flagging warnings if they return false.

`fail-if` and `warn-if` can be run on the command line using command-rose-macro or on-demand in the command-rose-config-edit GUI.

Note: Simple metadata settings such as `range` can be evaluated on-the-fly when a value changes. As `fail-if` and `warn-if` can take longer to evaluate they must be done on-demand in the command-rose-config-edit GUI or on the command line.

7.3.1 Syntax

The syntax is Pythonic, and relies on Jinja2 (http://jinja.pocoo.org/) to actually evaluate relationships between values, after some initial pre-processing.

You can reference setting values by using their IDs - for example:

```
fail-if=namelist:coffee=cup_volume < namelist:coffee=machine_output_volume;
```

You can also use `this` as a shorthand for the current (metadata section) ID - e.g.:

```
[namelist:coffee=daily_amount]
fail-if=this < namelist:coffee=daily_min or this >= namelist:coffee=daily_max;
```

There is also shorthand for arrays, which we'll demonstrate later.

Note that the `;` at the end is optional when we only have one expression (it's a delimiter), but it's better style to keep it.

7.3.2 Example

We'll use the example of a rocket launch.

Create a new application called `failif-warnif`:

```
mkdir -p ~/rose-tutorial
rose tutorial failif-warnif ~/rose-tutorial/failif-warnif
cd ~/rose-tutorial/failif-warnif
```

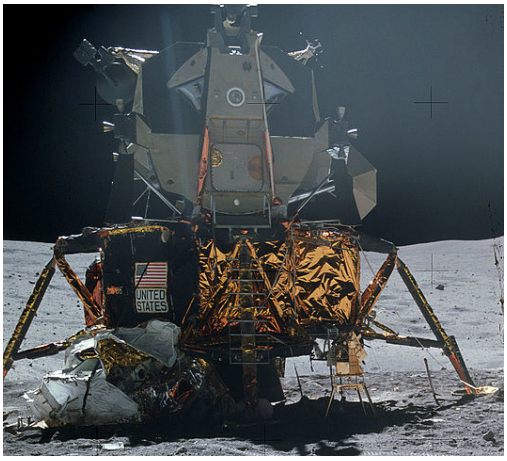
You will now have a new Rose app with a `rose-app.conf` that looks like this:

```
[command]
default=launch.exe

[env]
ORBITAL_SPEED_MS=1683.0

[file:rocket_settings.nl]
source=namelist:rocket

[namelist:rocket]
battery_levels=80, 60
total_weight_kg=4700.0
fuelless_weight_kg=2353.0
specific_impulse_s=311.0
```



This app configuration controls the liftoff of a particular rocket - in our case, the Lunar Module (Apollo Program spacecraft).

There is also metadata in the `meta/rose-meta.conf` file which provides the application inputs with descriptions, help text and type information.

Try running command-rose-config-edit in the app directory. You should be able to navigate between the pages and view the help and description for the settings.

7.3.3 `fail-if`

If the ratio of rocket fuel to total weight is too high, or the efficiency of the rocket (specific impulse) is too low, the Lunar Module will never make it off the Moon.

We want to be able to flag an error based on a combination of the rocket settings and the necessary orbital velocity (`env=ORBITAL_VELOCITY_MS`). We need to set some `fail-if` metadata on one of these settings - as it's evaluated on-demand, it doesn't matter which one we choose.

Open the `meta/rose-meta.conf` file in a text editor.

Add the following line to the metadata section `[namelist:rocket=total_weight_kg]`:

```
fail-if=this < namelist:rocket=fuelless_weight_kg * 2.7183**(env=ORBITAL_SPEED_MS /
↪ (9.8 * namelist:rocket=specific_impulse_s));
```

This states the relationship between these settings (a rearrangement of the Tsiolkovsky rocket equation (https://en.wikipedia.org/wiki/Tsiolkovsky_rocket_equation)). The rocket must have a sufficient ratio of fuel to rocket mass, with a sufficiently fast exhaust velocity (`=9.8 * namelist:rocket=specific_impulse_s`) to get to the orbital speed `env=ORBITAL_SPEED_MS`.

Save the metadata file and then reload the config editor metadata (*Metadata -> Refresh Metadata*).

You now need to ask Rose to evaluate the `fail-if` condition, as it's an on-demand process.

Either press the toolbar button *Check fail-if . . .* or click the menu item *Metadata → Check fail-if, warn-if*.

Hopefully, this should not flag any errors, as these are the Apollo mission parameters! A success message will appear in the bottom right-hand corner of the window.

Try adding a few more moonrocks. Add `1000` to the values of `total_weight_kg` and `fuelless_weight_kg`.

Re-run the check by clicking *Metadata → Check fail-if, warn-if*. An error dialog will appear, and the `total_weight_kg` setting will have an error flag.

However, neither of these are very informative, other than quoting the metadata.

Change the `fail-if` line to:

```
fail-if=this < namelist:rocket=fuelless_weight_kg * 2.7183**(env=ORBITAL_SPEED_MS /
↪ (9.8 * namelist:rocket=specific_impulse_s));  # Fuel mass ratio or specific␣
↪impulse too low to achieve orbit.
```

If you reload the metadata and run the check again, the error message will include the helpful text.

You can also check the `fail-if` metadata by running `rose macro --validate` or `rose macro -V` in a terminal, inside the app directory. Try saving the configuration in a failed state, and then run the command.

7.3.4 warn-if

The `warn-if` metadata setting is exactly the same as `fail-if`, but is used to report non-critical concerns.

Let's try adding something for `namelist:rocket=battery_levels`.

Open the metadata file `meta/rose-meta.conf` in a text editor, and add this line to the `[namelist:rocket=battery_levels]` section:

```
warn-if=namelist:rocket=battery_levels(1) < 75 or namelist:rocket=battery_
↪levels(2) < 75;
```

This uses a special syntax for referencing the individual array elements in `battery_levels`.

If the first array element value and/or the second array element value of `battery_levels` is less than 75% full, a warning will be produced when the check is run.

We already know the shorthand syntax `this`, so rephrase the metadata to:

```
warn-if=this(1) < 75 or this(2) < 75;
```

Save the metadata file and then reload the config editor metadata. Click *Metadata → Check fail-if, warn-if* - a warning should now appear for the `battery_levels` option.

For large arrays, it can sometimes be convenient to use whole-array operations - the `fail-if` and `warn-if` syntax includes `any()` and `all()`.

We can change the `warn-if` setting to:

```
warn-if=any(this < 75);
```

which will flag a warning if any `battery_levels` array element values are less than 75.

7.3.5 Multiple Expressions

In both `fail-if` and `warn-if`, expressions can be chained using the Python operator `or`, or you can separate them to give clearer error/warning messages. Using our `battery_levels` example again, change the setting to:

```
warn-if=any(this < 75);
       =all(this > 95);
```

This will produce a warning if any elements are less than 75, and a separate warning if all elements are greater than 95 (we don't want to cook the batteries!).

You can add separate helper messages for each expression:

```
warn-if=any(this < 75);    # Battery level low
       =all(this > 95);    # Don't over-charge!
```

Try adding the above lines to the metadata, saving and playing about with the array numbers in the config editor and re-running the `fail-if`/`warn-if` check.

Tip: For more information, see conf-meta.

7.4 Custom Macros

Rose macros are custom python modules which can perform checking beyond that which (e.g. `type`, `range`, `warn-if`, etc) can provide.

This tutorial covers the development of checking (**validator**), changing (**transformer**) and reporting (**reporter**) macros.

7.4.1 Warning

Macros should **only** be written if there is a genuine need that is not covered by other metadata - make sure you are familiar with conf-meta before you write your own (real-life) macros.

For example, `fail-if` and `warn-if` metadata options can perform complex inter-setting validation. See the *tutorial* (page 38) for details.

7.4.2 Purpose

Macros are used in Rose to report problems with a configuration, and to change it. Nearly all metadata mechanics (checking vs metadata settings, and changing - e.g. `trigger`) are performed within Rose by the Rose built-in macros.

Custom macros are user-defined, but follow exactly the same API - they are just in a different filespace location. They can be invoked via the command line (command-rose-macro) or from within the *Metadata* menu in the config editor.

7.4.3 Example

For these examples we will create an example app called `macro_tutorial_app` that could be part of a typical suite.

Create a directory for your suite app called `macro_tutorial_app`:

```
mkdir -p ~/rose-tutorial/macro_tutorial_app
```

Inside the `macro_tutorial_app` directory, create a `rose-app.conf` file and paste in the following contents:

```
[command]
default=echo "Hello $WORLD!"

[env]
WORLD=Earth
```

The metadata for the app lives under the `meta/` sub directory. Our new macro will live with the metadata.

For this example, we want to check the value of the option `env=WORLD` in our `macro_tutorial_app` application. Specifically, for this example, we want our macro to give us an error if the 'world' is too far away from Earth.

Create the directories `meta/lib/python/macros/` by running:

```
mkdir -p meta/lib/python/macros
```

Create an empty file called `rose-meta.conf` in the directory:

```
touch meta/rose-meta.conf
```

Create an empty file called `__init__.py` in the directory:

```
touch meta/lib/python/macros/__init__.py
```

Finally, create a file called `planet.py` in the directory:

```
touch meta/lib/python/macros/planet.py
```

Validator Macro

Open `planet.py` in a text editor and paste in the following code:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import re
import subprocess

import rose.macro


class PlanetChecker(rose.macro.MacroBase):

    """Checks option values that refer to planets."""

    opts_to_check = [("env", "WORLD")]

    def validate(self, config, meta_config=None):
        """Return a list of errors, if any."""
        for section, option in self.opts_to_check:
            node = config.get([section, option])
            if node is None or node.is_ignored():
                continue
            # Check the option value (node.value) here
        return self.reports
```

This is the bare bones of a Rose macro - a bit of Python that is a subclass of `rose.macro.MacroBase`. At the moment, it doesn't do anything.

We need to check the value of the option (`env=WORLD`) in our app configuration. To do this, we'll generate a list of allowed 'planet' choices that aren't too far away from Earth at the moment.

Call a method to get the choices by adding the line:

```
allowed_planets = self._get_allowed_planets()
```

at the top of the `validate` method, so it looks like this:

```
def validate(self, config, meta_config=None):
    """Return a list of errors, if any."""
    allowed_planets = self._get_allowed_planets()
```

Now add the method `_get_allowed_planets` to the class:

```
def _get_allowed_planets(self):
    # Retrieve planets less than a certain distance away.
    cmd_strings = ["curl", "-s",
                   "http://www.heavens-above.com/planetsummary.aspx"]
    p = subprocess.Popen(cmd_strings, stdout=subprocess.PIPE)
    text = p.communicate()[0]
    planets = re.findall("(\w+)</td>",
                         re.sub('(?s)^.*(tablehead.*?ascension).*$',
                                r"\1", text))
    distances = re.findall("([\d.]+)</td>",
                           re.sub('(?s)^.*(Range.*?Brightness).*$',
                                  r"\1", text))
    for planet, distance in zip(planets, distances):
        if float(distance) > 5.0:
            # The planet is more than 5 AU away.
            planets.remove(planet)
    planets += ["Earth"]  # Distance ~ 0
    return planets
```

This will give us a list of valid (nearby) solar system planets which our configuration option should be in. If it isn't, we need to send a message explaining the problem. Add:

```
error_text = "planet is too far away."
```

at the top of the class, like this:

```
class PlanetChecker(rose.macro.MacroBase):

    """Checks option values that refer to planets."""

    error_text = "planet is too far away."
    opts_to_check = [("env", "WORLD")]

    def validate(self, config, meta_config=None):
        """Return a list of errors, if any."""
        allowed_planets = self._get_allowed_planets()
```

Finally, we need to check if the configuration option is in the list, by replacing

```
# Check the option value (node.value) here
```

with:

```
if node.value not in allowed_planets:
    self.add_report(section, option, node.value, self.error_text)
```

The `self.add_report` call is invoked when the planet choice the user has made is not in the allowed planets. It adds the error information about the section and option (`env` and `WORLD`) to the `self.reports` list, which is returned to the rest of Rose to see if the macro reports any problems.

Your final macro should look like this:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import re
import subprocess

import rose.macro


class PlanetChecker(rose.macro.MacroBase):

    """Checks option values that refer to planets."""

    error_text = "planet is too far away."
    opts_to_check = [("env", "WORLD")]

    def validate(self, config, meta_config=None):
        """Return a list of errors, if any."""
        allowed_planets = self._get_allowed_planets()
        for section, option in self.opts_to_check:
            node = config.get([section, option])
            if node is None or node.is_ignored():
                continue
            if node.value not in allowed_planets:
                self.add_report(section, option, node.value, self.error_text)
        return self.reports

    def _get_allowed_planets(self):
        # Retrieve planets less than a certain distance away.
        cmd_strings = ["curl", "-s",
                       "http://www.heavens-above.com/planetsummary.aspx"]
        p = subprocess.Popen(cmd_strings, stdout=subprocess.PIPE)
        text = p.communicate()[0]
        planets = re.findall("(\w+)</td>",
                             re.sub(r'(?s)^.*(<thead.*?ascension).*$',
                                    r"\1", text))
        distances = re.findall("([\d.]+)</td>",
                               re.sub('(?s)^.*(Range.*?Brightness).*$',
                                      r"\1", text))
        for planet, distance in zip(planets, distances):
            if float(distance) > 5.0:
                # The planet is more than 5 AU away.
                planets.remove(planet)
        planets += ["Earth"]  # Distance ~ 0
        return planets
```

Results

Your validator macro is now ready to use.

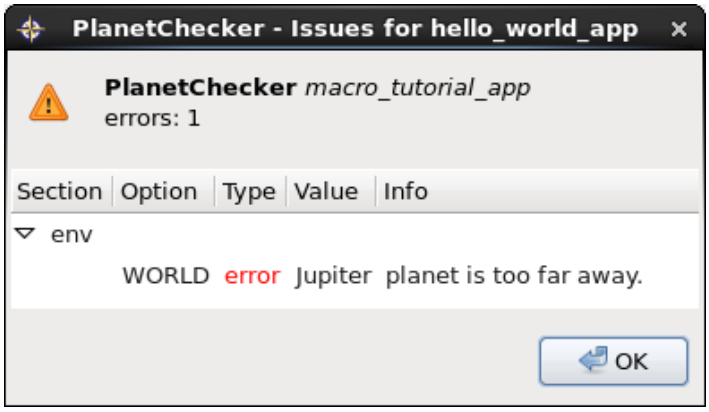
Run the config editor with the command:

```
rose edit
```

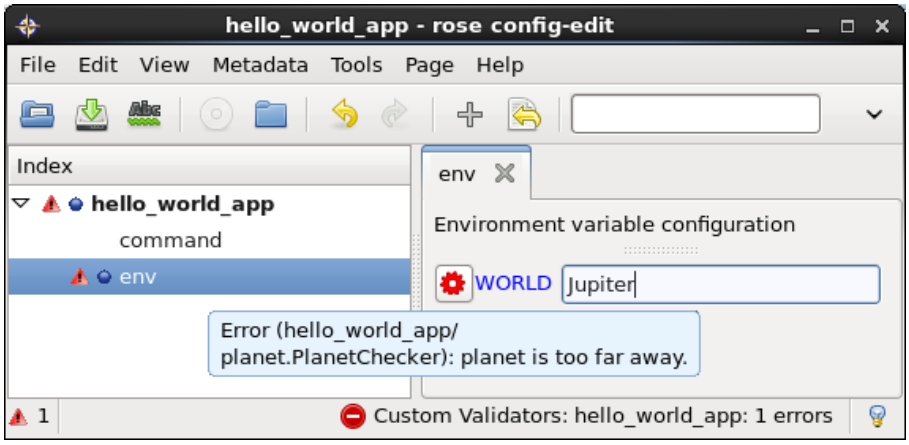
in the application directory. Navigate to the `env` page, and change the option `env=WORLD` to `Jupiter`.

To run the macro, select the menu *Metadata → macro_tutorial_app → planet.PlanetChecker.validate*.

It should either return an "OK" dialog, or give an error dialog like the one below depending on the current Earth-Jupiter distance.



If there is an error, the variable should display an error icon on the `env` page, which you can hover-over to get the error text as in the screenshot below. You can remove the error by fixing the value and re-running your macro.



Try changing the value of `env=WORLD` to other solar system planets and re-running the macro.

You can also run your macro from the command line:

```
rose macro planet.PlanetChecker
```

Transformer Macro

We'll now make a macro that changes the configuration. Our example will change the value of `env=WORLD` to something else.

Open `planet.py` in a text editor and append the following code:

```
class PlanetChanger(rose.macro.MacroBase):

    """Switch between planets."""

    change_text = '{0} to {1}'
    opts_to_change = [("env", "WORLD")]
    planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
               "Uranus", "Neptune", "Eris"]

    def transform(self, config, meta_config=None):
        """Transform configuration and return it with a list of changes."""
        for section, option in self.opts_to_change:
            node = config.get([section, option])
            # Do something to the configuration.
        return config, self.reports
```

This is another bare-bones macro class, although this time it supplies a `transform` method instead of a `validate` method.

You can see that it returns a configuration object (`config`) as well as `self.reports`. This means that you can modify the configuration e.g. by adding or deleting a variable and then returning the changed config object.

We need to add some code to make some changes to the configuration.

Replace the line:

```
# Do something to the configuration.
```

with:

```
if node is None or node.is_ignored():
    continue
old_planet = node.value
try:
    index = self.planets.index(old_planet)
except (IndexError, ValueError):
    new_planet = self.planets[0]
else:
    new_planet = self.planets[(index + 1) % len(self.planets)]
config.set([section, option], new_planet)
```

This changes the option `env=WORLD` to the next planet on the list. It will set it to the first planet on the list if it is something else. It will skip it if it is missing or ignored.

We also need to add a change message to flag what we've changed.

Beneath the line:

```
config.set([section, option], new_planet)
```

add the following two lines:

```
message = self.change_text.format(old_planet, new_planet)
self.add_report(section, option, new_planet, message)
```

This makes use of the template `self.change_text` at the top of the class. The message will be used to provide more information to the user about the change.

Your class should now look like this:

```
class PlanetChanger(rose.macro.MacroBase):

    """Switch between planets."""

    change_text = '{0} to {1}'
    opts_to_change = [("env", "WORLD")]
    planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
               "Uranus", "Neptune", "Eris"]

    def transform(self, config, meta_config=None):
        """Transform configuration and return it with a list of changes."""
        for section, option in self.opts_to_change:
            node = config.get([section, option])
            if node is None or node.is_ignored():
                continue
            old_planet = node.value
            try:
                index = self.planets.index(old_planet)
            except (IndexError, ValueError):
                new_planet = self.planets[0]
```

(continues on next page)

(continued from previous page)

```
            else:
                new_planet = self.planets[(index + 1) % len(self.planets)]
            config.set([section, option], new_planet)
            message = self.change_text.format(old_planet, new_planet)
            self.add_report(section, option, new_planet, message)
        return config, self.reports
```

Your transform macro is now ready to use.

You can run it from command-rose-config-edit via the menu *metadata* → *macro_tutorial_app* → *planet.PlanetChanger.transform*.

It should give a dialog explaining the changes it's made and asking for permission to apply them. If you click OK, the changes will be applied and the value of `env=WORLD` will be changed. You can Undo and Redo macro changes.

Try running the macro once or twice more to see it change the configuration.

You can also run your macro from the command line in the application directory by invoking `rose macro planet.PlanetChanger`.

Reporter Macro

Along with validator and transformer macros there are also reporter macros. These are used when you want to output information about a configuration but do not want to make any changes to it.

Next we will write a reporter macro which produces a horoscope entry based on the value of `env=WORLD`.

Open `planet.py` and paste in this text:

```
class PlanetReporter(rose.macro.MacroBase):

    """Creates a report on the value of env=WORLD."""

    GENERIC_HOROSCOPE_STATEMENTS = [
        'be cautious', 'remain indoors', 'expect the unexpected',
        'not walk under ladders', 'seek new opportunities']

    def report(self, config, meta_config=None):
        world_node = config.get(["env", "WORLD"])
        if world_node is None or world_node.is_ignored():
            return
        planet = world_node.value
        if planet.lower() == 'earth':
            print 'Please choose a planet other than Earth.'
            return
        constellation = self.get_planet_info(planet)
        if not constellation:
            print 'Could not find horoscope entry for {0}'.format(planet)
            return
        else:
            print (
                '{planet} is currently passing through {constellation}.\n'
                'You should {generic_message} today.'
            ).format(
                planet = planet,
                constellation = constellation,
                generic_message = random.choice(
                  self.GENERIC_HOROSCOPE_STATEMENTS)
            )

    def get_planet_info(self, planet_name):
```

(continues on next page)

(continued from previous page)

```
        cmd_strings = ["curl", "-s",
                       "http://www.heavens-above.com/planetsummary.aspx"]
        p = subprocess.Popen(cmd_strings, stdout=subprocess.PIPE)
        text = p.communicate()[0]
        planets = re.findall("(\w+)</td>",
                             re.sub(r'(?s)^.*(<thead.*?ascension).*$',
                                    r"\1", text))
        constellations = re.findall("(\w+)</a>",
                                re.sub('(?s)^.*(Constellation.*?Meridian).*$',
                                       r"\1", text))
        for planet, constellation in zip(planets, constellations):
            if planet.lower() == planet_name.lower():
                return constellation
        return None
```

You will need to add the following line with the other imports at the top of the file.

```
import random
```

Next run this macro from the command line by invoking:

```
rose macro planet.PlanetReporter
```

7.4.4 Macro Arguments

From time to time, we may want to change some macro settings. Rather than altering the macro each time or creating a separate macro for every possible setting, we can make use of Python keyword arguments.

We will alter the transformer macro to allow us to specify the name of the planet we want to use.

Open `planet.py` and alter the `PlanetChanger` class to look like this:

```
class PlanetChanger(rose.macro.MacroBase):

    """Switch between planets."""

    change_text = '{0} to {1}'
    opts_to_change = [("env", "WORLD")]
    planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
               "Uranus", "Neptune", "Eris"]

    def transform(self, config, meta_config=None, planet_name=None):
        """Transform configuration and return it with a list of changes."""
        for section, option in self.opts_to_change:
            node = config.get([section, option])
            if node is None or node.is_ignored():
                continue
            old_planet = node.value
            if planet_name is None:
                try:
                    index = self.planets.index(old_planet)
                except (IndexError, ValueError):
                    new_planet = self.planets[0]
                else:
                    new_planet = self.planets[(index + 1) % len(self.planets)]
            else:
                new_planet = planet_name
            config.set([section, option], new_planet)
            message = self.change_text.format(old_planet, new_planet)
```

(continues on next page)

(continued from previous page)

```
            self.add_report(section, option, new_planet, message)
        return config, self.reports
```

This adds the `planet_name` argument to the transform method with a default value of `None`. On running the macro it will give you the option to specify a value for `planet_name`. If you do, then that will be used as the new planet.

Save your changes and run the transformer macro either from the command line or command-rose-config-edit. You should be prompted to provide a value for `planet_name`. At the command line this will take the form of a prompt while in command-rose-config-edit you will be presented with a dialog to enter values in, with defaults already entered for you.

Specify a value to use for `planet_name` using a quoted string, e.g. `"Vulcan"` and accept the proposed changes. The `WORLD` variable should now be set to `Vulcan`. Check your configuration to confirm this.

7.4.5 Metadata Option

If a macro addresses particular sections, namespaces, or options, then it makes sense to write the relationship down in the metadata for the particular settings. You can do this using the `macro` metadata option.

For example, our validator and transformer macros above are both specific to `env=WORLD`. Open the file `macro_tutorial_app/meta/rose-meta.conf` in a text editor, and add the following lines

```
[env=WORLD]
macro=planet.PlanetChecker, planet.PlanetChanger
```

Close the config editor if it is still open, and open the app in the config editor again. The env page should now contain a dropdown menu at the top of the page for launching the two macros.

7.5 Optional Configurations

Optional configurations are configuration files which can add or overwrite the default configuration. They can be used with command-rose-app-run for Rose application configurations and command-rose-suite-run for Rose suite configurations.

7.5.1 Example

Create a new Rose app called `rose-opt-conf-tutorial`:

```
mkdir -p ~/rose-tutorial/rose-opt-conf-tutorial
cd ~/rose-tutorial/rose-opt-conf-tutorial
```

Create a `rose-app.conf` file with the following contents:

```
[command]
default=echo "I'd like to order a $FLAVOUR ice cream in a $CONE_TYPE" \
       ="with $TOPPING."

[env]
CONE_TYPE=regular-cone
FLAVOUR=vanilla
TOPPING=no toppings
```

Test the app by running:

```
rose app-run -q
```

You should see the following output:

```
I'd like to order a vanilla ice cream in a regular-cone with no toppings.
```

7.5.2 Adding Optional Configurations

Optional configurations are stored in the `opt` directory and are named the same as the default configuration file but with the name of the optional configuration before the `.conf` extension i.e:

```
app/
 |-- rose-app.conf
```

(continues on next page)

(continued from previous page)

```
`-- opt/
    `-- rose-app-<optional-configuration-name>.conf
```

Next we will create a new optional configuration for chocolate ice cream. The configuration will be called `chocolate`.

Create an `opt` directory containing a `rose-app-chocolate.conf` file containing the following configuration:

```
[env]
FLAVOUR=chocolate
```

Next we need to tell command-rose-app-run to use the `chocolate` optional configuration. We can do this in one of two ways:

1. Using the `--opt-conf-key` option.
2. Using the `ROSE_APP_OPT_CONF_KEYS` environment variable.

Run the app using the `chocolate` optional configuration:

```
rose app-run -q --opt-conf-key=chocolate
```

You should see the following output:

```
I'd like to order a chocolate ice cream in a regular-cone with no toppings.
```

The `chocolate` optional configuration has overwritten the `FLAVOUR` environment variable from the `rose-app.conf` file.

7.5.3 Using Multiple Optional Configurations

It is possible to use multiple optional configurations at the same time.

Create a new optional configuration called `flake` containing the following configuration:

```
[env]
TOPPING=one chocolate flake
```

Run the app using both the `chocolate` and `flake` optional configurations:

```
rose app-run -q --opt-conf-key=chocolate --opt-conf-key=flake
```

The `FLAVOUR` environment variable will be overwritten by the `chocolate` configuration and the `TOPPING` variable by the `flake` configuration.

Next create a new optional configuration called `fudge-sundae` containing the following lines:

```
[env]
FLAVOUR=fudge
CONE_TYPE=tub
TOPPINGS=nuts
```

Run the app using both the `chocolate` and `fudge-sundae` optional configurations:

```
rose app-run -q --opt-conf-key=fudge-sundae --opt-conf-key=chocolate
```

You should see the following:

```
I'd like to order a chocolate icecream in a tub with nuts.
```

The `chocolate` configuration has overwritten the `FLAVOUR` environment variable from the `fudge sundae` configuration. This is because optional configurations as applied first to last so in this case the `chocolate` configuration was loaded last.

To see how the optional configurations would be applied use the command-rose-config command providing the configuration files in the order they would be loaded:

```
rose config --file rose-app.conf --file opt/rose-app-fudge-sundae --file chocolate
```

You should see:

```
[command]
default=echo "I'd like to order a $FLAVOUR icecream in a $CONE_TYPE" \
       ="with $TOPPING toppings"

[env]
CONE_TYPE=tub
FLAVOUR=chocolate
TOPPING=nuts
```

Note: Optional configurations specified using the `ROSE_APP_OPT_CONF_KEYS` environment variable are loaded before those specified using the `--opt-conf-key` command line option.

7.5.4 Using Optional Configurations By Default

Optional configurations can be switched on by default using the `opt` setting.

Add the following line at the top of the `rose-app.conf` file:

```
opts=chocolate
```

Now the `chocolate` optional configuration will *always* be turned on. For this reason its generally better to use the `--opt-conf-key` setting or `ROSE_APP_OPT_CONF_KEYS` environment variable instead.

7.5.5 Other Optional Configurations

All Rose configurations can have optional configurations, not just application configurations.

- Suites can have optional configurations that override `rose-suite.conf` settings, controlled through command-rose-suite-run. Optional suite configurations can be used either using the `--opt-conf-key` option with command-rose-suite-run or the `ROSE_SUITE_OPT_CONF_KEYS` environment variable.
- Metadata configurations can also have optional configurations, typically included via the `rose-app.conf|opts` top-level setting.

7.6 Polling

Polling allows you to check for some condition to be met prior to running the main command in an app without the need for additional entries in the dependencies graph.

For example, you might want to run a polling command to check for the existence of a particular file before running the main command which requires said file.

7.6.1 Example

Create a new Rose suite configuration:

```
mkdir -p ~/rose-tutorial/polling
cd ~/rose-tutorial/polling
```

Create a blank `rose-suite.conf` and a `suite.rc` file that looks like this:

```
[cylc]
    UTC mode = True # Ignore DST
[scheduling]
    [[dependencies]]
        graph = """compose_letter => send_letter
                   bob => read_letter"""
```

This is a simple suite which consists of the following:

- A `compose_letter` task.
- A `send_letter` task which is run once the letter is composed.
- A `bob` task which we will be using to poll with.
- A `read_letter` task which will run once the polling task is complete.

It will need some runtime. Add the following to your `suite.rc` file:

```
[runtime]
    [[root]]
        script = sleep 10
    [[compose_letter]]
        script = sleep 5; echo 'writing a letter to Bob...'
    [[send_letter]]
        env-script = eval `rose task-env`
        script = """
                 sleep 5
                 echo 'Hello Bob' > $ROSE_DATA/letter.txt
                 sleep 10
                 """

    [[bob]]
        script = rose task-run
    [[read_letter]]
        env-script = eval `rose task-env`
        script = sleep 5; cat $ROSE_DATA/letter.txt
        post-script = rm $ROSE_DATA/letter.txt
```

7.6.2 Adding Polling

In the suite directory create an `app` directory.

In the `app` directory create a directory called `bob`.

In the newly-created `bob` directory, create a `rose-app.conf` file.

Edit the `rose-app.conf` file to look like this:

```
[poll]
delays=10*PT5S
test=test -e $ROSE_DATA/letter.txt

[command]
default=echo 'Ooh, a letter!'
```

We now have an app that does the following:

- Has a polling `test` that checks for the existence of a file.
- Polls up to 10 times with 5 second delays between each attempt.
- Prints a message once the polling test succeeds.

Note: The ordering of the `[poll]` and `[command]` sections is not important. In practice, it may be preferable to have the `[command]` section at the top as that should contain the main command(s) being run by the app.

Save your changes and run the suite using command-rose-suite-run.

The suite should now run.

Notice that `bob` finishes and triggers `read_letter` before `send_letter` has completed. This is because the polling condition has been met, allowing the main command in `bob` to be run.

7.6.3 Improving The Polling

At present we have specified our own routine for testing for the existence of a particular file using the `test` option. However, Rose provides a simpler method for doing this.

Edit the `rose-app.conf` in your `bob` app to look like the following:

```
[poll]
delays=10*PT5S
all-files=$ROSE_DATA/letter.txt

[command]
default=echo 'Ooh, a letter!'
```

Polling is now making use of the `all-files` option, which allows you to specify a list of files to check the existence of. Save your changes and run the suite to confirm it still works.

7.6.4 Available Polling Types

Test and all-files are just two of the available polling options:

`all-files` Tests if all of the files in a list exist.

`any-files` Tests if any of the files in a list exist.

`file-test` Changes the test used to evaluate the `any-files` and `all-files` lists to a shell script to be run on each file (e.g. `grep`). Passes if the command exits with a zero return code.

`test` Tests using a shell script, passes if the command exits with a zero return code. *Note this is separate from the* `all-files`, `any-files` *testing logic.*

Tip: For more details see Rose Applications.

7.6.5 Possible Uses For Polling

Depending on your needs, possible uses for polling might include:

- Checking for required output from a long-running task rather than waiting for the task to complete.
- Monitoring output from another suite.
- Checking if a file has required content before using it.

7.7 Rose Arch

`rose_arch` is a built-in Rose app that provides a generic solution to the archiving of suite files.

Good Practice

Only archive the minimum files needed at each cycle of your suite. Run the archiving task before any housekeeping in the graph.

7.7.1 Example

Create a new Rose suite configuration:

```
mkdir -p ~/rose-tutorial/rose-arch-tutorial
```

Create a blank `rose-suite.conf` and a `suite.rc` file that looks like this:

```
[cylc]
    UTC mode = True # Ignore DST
    [[events]]
        abort on timeout = True
        timeout = PT1H
[scheduling]
    [[dependencies]]
        graph = make_files => archive_files_rsync => archive_files_scp
[runtime]
    [[root]]
        env-script = eval $(rose task-env)
        script = rose task-run

    [[make_files]]
        script = """
            echo 'zip' >> $ROSE_DATAC/file_zip
            echo 'solo' >> $ROSE_DATAC/file_solo
            echo 'list1' >> $ROSE_DATA/file_list1
            echo 'list2' >> $ROSE_DATA/file_list2
            echo 'list3' >> $ROSE_DATA/file_list3
            mkdir -p $ROSE_DATA/ARCHIVING || true
            mkdir -p $ROSE_DATA/ARCHIVING/rename || true
        """
    [[archive_files_rsync]]
    [[archive_files_scp]]
```

In the suite directory create an `app/` directory:

```
mkdir app
```

In the app directory create an `archive_files_rsync/` directory:

```
cd app
mkdir archive_files_rsync
```

In the `app/archive_files_rsync/` directory create a `rose-app.conf` file. This example uses `vi`, but please use your editor of choice:

```
cd archive_files_rsync
vi rose-app.conf
```

Paste in the following lines:

```
mode=rose_arch

[env]
ARCH_TARGET=$ROSE_DATA/ARCHIVING

[arch]
command-format=rsync -a %(sources)s %(target)s
source-prefix=$ROSE_DATAC/
target-prefix=$ARCH_TARGET/
update-check=mtime+size

[arch:solo.file]
source=file_solo

[arch:files]
source=file_list1 file_list3
source-prefix=$ROSE_DATA/

[arch:dir]
source=file*
source-prefix=$ROSE_DATA/

[arch:file_zipped.tar]
source=file_zip
```

Move to the `app/` directory:

```
cd ..
ls
```

The following should be returned:

```
archive_files_rsync
```

Create an `archive_files_scp/` directory:

```
mkdir archive_files_scp
```

In the `archive_files_scp/` directory create a `rose-app.conf` file. This example uses `vi`, but please use your editor of choice:

```
cd archive_files_scp
vi rose-app.conf
```

Paste in the following lines:

```
mode=rose_arch

[env]
ARCH_TARGET=$ROSE_DATA/ARCHIVING

[arch]
command-format=scp %(sources)s %(target)s
source-prefix=$ROSE_DATA/
target-prefix=$ARCH_TARGET/
update-check=mtime+size

[arch:rename/]
rename-format=%(cycle)s_%(tag)s_%(name)s
rename-parser=^.*list(?P<tag>.*)$
source=file_list?
```

7.7.2 Description

You have now created a suite that defines three tasks:

make_files Sets up the files and ARCHIVING/ directory for archive_files_rsync/ and archive_files_scp/ to "archive", move, data to.

archive_files_rsync "Archives" (rsync's) files to the ARCHIVING/ folder in the $ROSE_DATA/ directory.

archive_files_scp "Archives" (scp's) the renamed files and moves them to the ARCHIVING/ folder in the $ROSE_DATA/ directory.

Save your changes and run the suite:

```
rose suite-run
```

View the suite output using command-rose-suite-log and inspect the output of the make_files, archive_files_rsync and archive_files_scp tasks.

7.7.3 Results Of "Archiving"

Change to the $ROSE_DATA/ARCHIVING/ directory of the suite i.e:

```
cd ~/cylc-run/<name>/share/data/ARCHIVING/
```

List the directory by typing:

```
ls
```

You should see the following returned:

```
dir  file_zipped.tar  files  rename  solo.file
```

Change directory to files/ and list the files:

```
cd files
ls
```

The following should be returned:

```
file_list1  file_list3
```

Change directory to ARCHIVING/dir/ and list the files:

```
cd ..
cd dir
ls
```

The following should be returned:

```
file_list1  file_list2 file_list3
```

Note: These were all of the files in the $ROSE_DATA/ directory.

Change directory to ARCHIVING/rename/ and list the files:

```
cd ..
cd rename
ls
```

The following should be returned:

```
1_1_file_list1 1_2_file_list2 1_3_file_list3
```

These are the renamed files.

Most users will have their own system or location that they wish to archive their data to. Here the example shown uses rsync (https://linux.die.net/man/1/rsync) and scp (https://www.lifewire.com/rcp-scp-ftp-commands-for-copying-files-3971107). Please refer your own site specific archiving solutions and seek site specific advice.

7.7.4 Arch Settings

Some settings that can be used are described below. See the rose_arch documentation for more information:

Above `.tar` was used to compress the file. However, `compress=gzip` can also be used. Note either of these commands can be used to compress a file or a folder/directory.

In the above example a regular expression 'reg exp' was used by the `rename-parser`, for example, `^.*list(?P<tag>.*)$`, where:

- `^` = start of a string.
- `$` = end of a string.
- `.` = any character.
- `*` = greedy (https://stackoverflow.com/questions/2301285/what-do-lazy-and-greedy-mean-in-the-context-of-regular-expressions) (all).
- `?P<NAME>` = named group.

Note: `rose arch` uses the Python flavor (https://docs.python.org/2/howto/regex.html) for regular expressions.

In the above example source was used to accept a list of glob patterns. For example, `file_list?` was used where the `?` relates to one unknown character.

Note: These examples are just some possible examples and not a full list.

As well as `rose_arch[arch]` and `[arch:TARGET]` other options can be provided to the app, for example:

`[env]` Can be defined near the top of the app to allow an environment variable to be available to the `[arch:]` commands in the app.

Also see `rose-app.conf[env]` and the suite example above.

`[poll]` Polling can be defined, and is often near the bottom of the app. This will allow the app to poll with a defined delay, e.g. `rose-app.conf[poll]delays=5`.

`[file:TARGET]` This option allows the user to, for example, make the directory `TARGET`, e.g. `*[file:TARGET]mode=mkdir`.

For more information, see the rose_arch documentation.

7.8 Rose Bunch

`rose_bunch` is a built-in Rose app which allows multiple variants of a command to be run under a single job.

7.8.1 Purpose

Often, we want to run many instances of a command that differ only slightly from each other at the same time - an example would be where a command is run repeatedly with only its arguments changing.

Rather than creating multiple apps or *optional configs* (page 49) to change the way a command is to be run, we can instead use the built-in `rose_bunch` application to run multiple command variants, in parallel, under a single job as defined by an application configuration.

Note, however, that for "embarrassingly parallel" code it would be better to alter the code rather than use `rose_bunch` to handle this for you.

> **Warning:** It is important to note that when running your `rose_bunch` app under load balancing systems such as PBS or Slurm, you will need to set resource requests to reflect the resources required by running multiple commands at once.
>
> For example, if a single command would require 1GB memory and the app is configured to run up to 4 commands at once then 4GB of memory should be requested.

7.8.2 Example

In this example we are going to create a suite that simulates the handling of landing planes at an airport. For a given plane the process of landing and unloading is the same: land, taxi to the terminal, unload passengers and get clear. We can refer to this as the "landing" routine. What differs between landings is the plane type, number of passengers carried and the resulting timings for each stage of the landing process.

Create a new Rose suite configuration:

```
mkdir -p ~/rose-tutorial/rose-bunch
cd ~/rose-tutorial/rose-bunch
```

Create a blank `rose-suite.conf` and a `suite.rc` file that looks like this:

```
[cylc]
    UTC mode = True # Ignore DST
[scheduling]
    [[dependencies]]
        graph = lander
[runtime]
    [[root]]
        script = rose task-run
    [[lander]]
```

In the suite directory create an `app/` directory:

```
mkdir app
```

In the app directory create a `lander/` directory:

```
cd app
mkdir lander
```

In the `app/lander/` directory create a `rose-app.conf` file using your editor of choice and paste the following lines into it:

```
mode=rose_bunch

[bunch]
command-format=land %(class)s %(passengers)s
```

(continues on next page)

(continued from previous page)

```
[bunch-args]
class=airbus concorde airbus cessna
passengers=40 20 30 2
```

This configuration will run a `rose_bunch` task that calls multiple instances of the `land` command, supplying arguments to each instance from the `class` and `passengers` entries under `rose_bunch[bunch-args]`.

In the `app/lander/` directory create a `bin/` directory:

```
mkdir bin
```

Using your editor of choice, create a file named `land` under the `bin` directory and paste in these lines:

```
#!/bin/bash

CLASS=$1
PASSENGERS=$2

# Get settings
case $CLASS in
    airbus) LANDTIME=30; UNLOADRATE=8;;
    cessna) LANDTIME=20; UNLOADRATE=2;;
    concorde) LANDTIME=10; UNLOADRATE=4;;
esac

echo "[ $(rose date) ] $CLASS carrying $PASSENGERS passengers incoming"

# Land plane
echo "[ $(rose date) ] Approaching runway"
sleep $LANDTIME
echo "[ $(rose date) ] On the tarmac"

# Unload passengers
sleep $(($PASSENGERS / $UNLOADRATE))
echo "[ $(rose date) ] Unloaded"

# Clear terminal
sleep 10
echo "[ $(rose date) ] Clear of terminal"
```

This script captures the landing routine and expects two arguments: the plane type (its class) and the number of passengers it is carrying.

Finally, make the new `land` file executable by navigating into the `bin` directory of the lander app and running:

```
chmod +x land
```

Navigate to the top directory of your suite (where the `suite.rc` and `rose-suite.conf` files can be found) and run command-rose-suite-run.

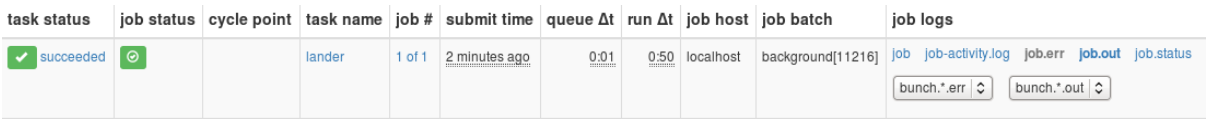
Your suite should run, launch the Cylc GUI and successfully run the `lander` app.

Once the suite has finished running and has shutdown, open Rose Bush to view its output (note that you can close the Cylc GUI at this point):

```
rose suite-log
```

Note: You can quickly get to the relevant page by running command-rose-suite-log from within the suite directory.

In the Rose Bush jobs page for your suite you should be presented with a page containing a single row for the `lander` task, from which you can access its output. In that row you should see something like this:



In the Rose Bush entry you should see that the usual links are present for the task such as `job.out`, `job.status` etc. with the addition of two drop-down boxes: one for `bunch.*.err` and one for `bunch.*.out`. Rather than mixing the outputs from the multiple command invocations being run at once, `rose_bunch` directs their output to individual output files. So, for example, the output from running the command with the first set of parameters can be found in the `bunch.0.out` file, the second set in the `bunch.1.out` file etc. Examine these output files now to confirm that all four of the args combinations have been run and produced output.

7.8.3 Naming Invocations

While the different invocations of the command have their own output directed to indexed files, it can sometimes be difficult to quickly identify which file to look in for output. To aid this, `rose_bunch` supports naming command instances via the `rose_bunch[bunch]names=` option.

Open your app config (under `app/lander/rose-app.conf`) and add the following line under the `rose_bunch[bunch]` section:

```
names=BA123 Emirates345 BA007 PC456
```

Re-run your suite and, once it has finished, open up Rose Bush and examine the job listing. In the drop-down `bunch.*.err` and `bunch.*.out` boxes you should now see entries for the names you've configured rather than the `bunch.0.out ... bunch.3.out` entries previously present.

7.8.4 Limiting Concurrent Invocations

In some situations we may need to limit the number of concurrently running command invocations - often as a result of resource limitations. Rather than batching up jobs into sets of *N* simultaneously running commands, `rose_bunch` apps can be configured to run as many commands as possible within some limit i.e. while *N* commands are running, if one of them finishes, don't wait for the remaining *N*-1 jobs to finish before running the (*N*+1)th one.

In the case of our simulated airport we will pretend we only have two runways available at a time on which our planes can land. As such we need to limit the number of planes landing. We do this using the `rose_bunch[bunch]pool-size=` configuration option of the `rose_bunch` app.

Open your app config (under `app/lander/rose-app.conf`) and add the following line to the `rose_bunch[bunch]` section:

```
pool-size=2
```

Run your suite again. Notice that this time round it takes longer for the task to run as it has been limited in the number of command variants it can run simultaneously. You can see the individual commands being started by viewing the task stdout in the Cylc GUI by right-clicking on the task and selecting *View* then *job stdout*. As an example, when the `BA007` invocation starts running you should see the line:

```
[INFO] BA007: added to pool
```

appear in the job output after a while whereas, when running without a `rose_bunch[bunch]pool-size`, the line will appear pretty quickly.

7.8.5 Summary

In this tutorial we have learnt how to configure a `rose_bunch` app to run a set of command variants under one job. We have learnt how to name the individual variants for convenience in examining the logs and how to limit the number of concurrently running commands.

Further options are listed in the `rose_bunch` documentation. These include configuring how to proceed following failure of an individual command invocation (`rose_bunch[bunch]fail-mode=`), automatically generating *N* command instances and enabling/disabling the app's incremental mode.

7.9 Rose Stem

Rose Stem is a testing system for use with Rose. It provides a user-friendly way of defining source trees and tasks on the command line which are then passed by Rose Stem to the suite as Jinja2 variables.

> **Warning:** Rose Stem requires the use of FCM (https://metomi.github.io/fcm/doc/) as it requires some of the version control information.

7.9.1 Motivation

Why do we test code?

Most people would answer something along the lines of "so we know it works".

However, this is really asking two related but separate questions.

1. Does the code do what I meant it to do?
2. Does the code do anything I didn't mean it to do?

Answering the first question may involve writing a bespoke test and checking the results. The second question can at least partially be answered by using an automated testing system which runs predefined tasks and presents the answers. Rose Stem is a system for doing this.

N.B. When writing tests for new code, they should be added to the testing system so that future developers can be confident that they haven't broken the new functionality.

7.9.2 Rose Stem

There are two components in Rose Stem:

command-rose-stem The command line tool which executes an appropriate suite.

`rose_ana` A Rose built-in application which can compare the result of a task against a control.

We will describe each in turn. It is intended that a test suite lives alongside the code in the same version-controlled project, which should encourage developers to update the test suite when they update the code. This means that the test suite will always be a valid test of the code it is accompanying.

7.9.3 Running A Suite With command-rose-stem

The `rose stem` command is essentially a wrapper to command-rose-suite-run, which accepts some additional arguments and converts them to Jinja2 variables which the suite can interpret.

These arguments are:

`--source` Specifies a source tree to include in a suite.

`--group` Specifies a group of tasks to run.

A group is a set of Rose tasks which together test a certain configuration of a program.

7.9.4 The `--source` Argument

The source argument provides a set of Jinja2 variables which can then be included in any compilation tasks in a suite. You can specify multiple `--source` arguments on the command line. For example:

```
rose stem --source=/path/to/workingcopy --source=fcm:other_project_tr@head
```

Each source tree is associated with a project (via an `fcm` command) when command-rose-stem is run on the command line. This project name is then used in the construction of the Jinja2 variable names.

Each project has a Jinja2 variable `SOURCE_FOO` where `FOO` is the project name. This contains a space-separated list of all sourcetrees belonging to that project, which can then be given to an appropriate build task in the suite so it builds those source trees.

Similarly, a `HOST_SOURCE_FOO` variable is also provided. This is identical to `SOURCE_FOO` except any working copies have the local hostname prepended. This is to assist building on remote machines.

The first source specified must be a working copy which contains the Rose Stem suite. The suite is expected to be in a subdirectory named `rose-stem` off the top of the working copy. This source is used to generate three additional variables:

`SOURCE_FOO_BASE` The base directory of the project

`HOST_SOURCE_FOO_BASE` The base directory of the project with the hostname prepended if it is a working copy

`SOURCE_FOO_REV` The revision of the project (if any)

These settings override the variables in the `rose-suite.conf` file.

These should allow the use of configuration files which control the build process inside the working copy, e.g. you can refer to:

```
{{HOST_SOURCE_FOO_BASE}}/fcm-make/configs/machine.cfg{{SOURCE_FOO_REV}}
```

If you omit the source argument, Rose Stem defaults to assuming that the current directory is part of the working copy which should be added as a source tree:

```
rose stem --source=.
```

The project to which a source tree belongs is normally automatically determined using FCM (https://metomi.github.io/fcm/doc/) commands. However, in the case where the source tree is not a valid FCM URL, or where you wish to assign it to another project, you can specify this using the `--source` argument:

```
rose stem --source=foo=/path/to/source
```

assigns the URL `/path/to/source` to the foo project, so the variables `SOURCE_FOO` and `SOURCE_FOO_BASE` will be set to `/path/to/source`.

7.9.5 The `--group` Argument

The group argument is used to provide a Pythonic list of groups in the variable `RUN_NAMES` which can then be looped over in a suite to switch sets of tasks on and off.

Each `--group` argument adds another group to the list. For example:

```
rose stem --group=mygroup --group=myothergroup
```

runs two groups named `mygroup` and `myothergroup` with the current working copy. The suite will then interpret these into a set of tasks which build with the given source tree(s), run the program, and compare the output.

7.9.6 The `--task` Argument

The task argument is provided as a synonym for `--group`. Depending on how exactly the Rose Stem suite works users may find one of these arguments more intuitive to use than the other.

7.9.7 Comparing Output With `rose_ana`

Any task beginning with `rose_ana_` will be interpreted by Rose as a Rose Ana task, and run through the `rose_ana` built-in application.

A Rose Ana `rose-app.conf` file contains a series of blocks; each one describing a different analysis task to perform. A common task which Rose Ana is used for is to compare output contained in different files (e.g. from a new test versus previous output from a control). The analysis modules which provide these tasks are flexible and able to be provided by the user; however there is one built-in module inside Rose Ana itself.

An example based on the built-in `grepper` module:

```
[ana:grepper.FilePattern(Compare data from myfile)]
pattern='data value:(\d+)'
files=/data/kgo/myfile
     =../run.1/myfile
```

This tells Rose Ana to scan the contents of the file `../run.1/myfile` (which is relative to the Rose Ana task's work directory) and the contents of `/data/kgo/myfile` for the specified regular expression. Since the pattern contains a group (in parentheses) so it is the contents of this group which will be compared between the two files. The `grepper.FilePattern` analysis task can optionally be given a "tolerance" option for matching numeric values, but without it the matching is expected to be exact. If the pattern or group contents do not match the task will return a failure.

As well as sections defining analysis tasks, Rose Ana apps allow for one additional section for storing global configuration settings for the app. Just like the tasks themselves these options and their effects are dependent on which analysis tasks are used by the app.

Therefore we will here present an example using the built-in `grepper` class. An app may begin with a section like this:

```
[ana:config]
grepper-report-limit=5
skip-if-all-files-missing=.true.
```

Each of these modifies the behaviour of `grepper`. The first option suppresses printed output for each analysis task once the specified number of lines have been printed (in this case 5 lines). The second option causes Rose Ana to skip any `grepper` tasks which compare files in the case that both files do not exist.

Note: Any options given to this section may instead be specified in the `rose.conf[rose-ana]` section of the user or site configuration. In the case that the same configuration option appears in both locations the one contained in the app file will take precedence.

It is possible to add additional analysis modules to Rose Ana by placing an appropriately formatted python file in one of the following places (in order of precedence):

1. The `ana` sub-directory of the Rose Ana application.
2. The `ana` sub-directory of the suite.
3. Any other directory which is accessible to the process running Rose Ana and is specified in the `rose.conf[rose-ana]method-path` variable.

The only analysis module provided with Rose is `rose.apps.ana_builtin.grepper`, it provides the following analysis tasks and options:

class rose.apps.ana_builtin.grepper.**SingleCommandStatus**(*parent_app*, *task_options*)

Run a shell command, passing or failing depending on the exit status of that command.

Options:

files (optional): A newline-separated list of filenames which may appear in the command.

command: The command to run; if it contains Python style format specifiers these will be expanded using the list of files above (if provided).

kgo_file: If the list of files above was provided gives the (0-based) index of the file holding the "kgo" or "control" output for use with the comparisons database (if active).

class rose.apps.ana_builtin.grepper.**SingleCommandPattern**(*parent_app*, *task_options*)

Run a single command and then pass/fail depending on the presence of a particular expression in that command's standard output.

Options:

files (optional): Same as previous task. command - same as previous task.

kgo_file: Same as previous task.

pattern: The regular expression to search for in the stdout from the command.

class rose.apps.ana_builtin.grepper.**FilePattern**(*parent_app*, *task_options*)

Check for occurrences of a particular expression or value within the contents of two or more files.

Options:

files (optional): Same as previous tasks.

kgo_file: Same as previous tasks.

pattern: The regular expression to search for in the files. The expression should include one or more capture groups; each of these will be compared between the files any time the pattern occurs.

tolerance (optional): By default the above comparisons will be compared exactly, but if this argument is specified they will be converted to float values and compared according to the given tolerance. If this tolerance ends in % it will be interpreted as a relative tolerance (otherwise absolute).

class rose.apps.ana_builtin.grepper.**FileCommandPattern**(*parent_app*, *task_options*)

Check for occurrences of a particular expression or value in the standard output from a command applied to two or more files.

Options:

files (optional): Same as previous tasks.

kgo_file: Same as previous tasks.

pattern: Same as previous tasks.

tolerance (optional): Same as previous tasks.

command: The command to run; it should contain a Python style format specifier to be expanded using the list of files above.

The following options can be specified in:

- `rose.conf[rose-ana]`
- `rose_ana[ana:config]`

Options

grepper-report-limit: A numerical value giving the maximum number of informational output lines to print for each comparison. This is intended for cases where for example a pattern-matching comparison is expected to match many thousands of occurrences in the given files; it may not be desirable to print the results of every comparison. After the given number of lines are printed a special message indicating that the rest of the output is truncated will be produced.

skip-if-all-files-missing: Can be set to `.true.` or `.false.`; if active, any comparison done on files by `grepper` will be skipped if all of those files are non-existent. In this case the task will return as "skipped" rather than passed/failed.

The format for analysis modules themselves is relatively simple; the easiest route to understanding how they should be arranged is likely to look at the built-in `grepper` module. But the key concepts are as follows. To be recognised as a valid analysis module, the Python file must contain at least one class which inherits and extends *rose.apps.rose_ana.AnalysisTask* (page 66):

class rose.apps.rose_ana.**AnalysisTask**(*parent_app*, *task_options*)

Base class for an analysis task.

All custom user tasks should inherit from this class and override the `run_analysis` method to perform whatever analysis is required.

This class provides the following attributes:

`self.`**config**
A dictionary containing any Rose Ana configuration options.

`self.`**reporter**
A reference to the `rose.reporter.Reporter` instance used by the parent app (for printing to stderr/stdout).

`self.`**kgo_db**
A reference to the KGO database object created by the parent app (for adding entries to the database).

`self.`**popen**
A reference to the `rose.popen.RosePopener` instance used by the parent app (for spawning subprocesses).

For example:

```
from rose.apps.rose_ana import AnalysisTask

class CustomAnalysisTask(AnalysisTask):
    """My new custom analysis task."""
    def run_analysis(self):
        print self.options  # Dictionary of options (see next slide)
        if self.options["option1"] == "5":
            self.passed = True
```

Assuming the above was saved in a file called `custom.py` and placed into a folder suitable for analysis modules this would allow a Rose Ana application to specify:

```
[ana:custom.CustomAnalysisTask(Example rose-ana test)]
option1 = 5
option2 = test of Rose Ana
option3 = .true.
```

Note: The custom part of the filename appears at the start of the `ana` entry, followed by the name of the desired class (in the style of Python's own namespacing). All options specified by the app-task will be processed by Rose Ana into a dictionary and attached to the running analysis class instance as the options attribute. Hopefully you can see that in this case the task would pass because `option1` is set to 5 as required by the class.

7.9.8 The Rose Ana Comparison Database

In addition to performing the comparisons each of the Rose Ana tasks in the suite can be configured to append some key details about any comparisons performed to an sqlite database kept in the suite's log directory (at `log/rose-ana-comparisons.db`).

This is intended to provide a quick means to interrogate the suite for information about the status of any comparisons it has performed. There are 2 tables present in the suite which contain the following:

tasks (TABLE) The intention of this table is to detect if any Rose Ana tasks have failed unexpectedly (or are still running).

Contains an entry for each Rose Ana task, using the following columns:

task_name (TEXT) The exact name of the Rose Ana task.

completed (INT) Set to 1 when the task starts performing its comparisons then updated to 0 when the task has completed

---

Note: Task success is not related to the success/failed state of the comparisons).

---

comparisons (TABLE) The intention of this table is to provide a record of which files were compared by which tasks, how they were compared and what the result of the comparison was.

Contains an entry for each individual comparison from every Rose Ana task, using the following columns:

comp_task (TEXT) The comparison task name - by convention this is usually the comparison section name from the app definition (including the part inside the brackets).

kgo_file (TEXT) The full path to the file specified as the KGO file in the app definition.

suite_file (TEXT) The full path to the file specified as the active test output in the app definition.

status (TEXT) The status of the task (one of " OK ", "FAIL" or "WARN"). comparison (TEXT) Additional details which may be provided about the comparison.

The database is entirely optional; by default is will not be produced; if it is required it can be activated by setting `rose.conf[rose-ana]kgo-database=.true.`.

Note: The system does not provide any direct methods for working with or interrogating the database - since there could be various reasons for doing so, and there may be other suite-design factors to consider. Users are therefore expected to provide this functionality separately based on their specific needs.

7.9.9 Summary

From within a working copy, running `rose stem` is simple. Just run:

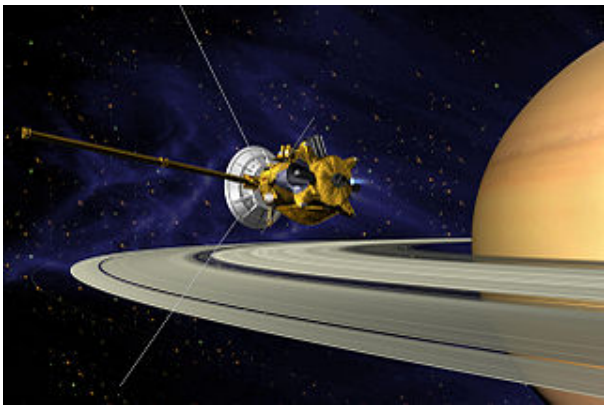
```
rose stem --group=groupname
```

replacing the groupname with the desired task. Rose Stem should then automatically pick up the working copy and run the requested tests on it.

Next see the *Rose Stem Tutorial* (page 68)

7.10 Rose Stem Tutorial

Warning: Before proceeding you should already be familiar with the *Rose Stem* (page 62) section.



This tutorial will walk you through creating a simple example of the Rose Stem testing system which will involve piloting a spaceship through space.

7.10.1 Getting Started

We will start the Rose Stem tutorial by setting up an FCM (https://metomi.github.io/fcm/doc/) repository called `SPACESHIP` to store the code and test suite in.

Usually you would add a Rose Stem suite to an existing repository with the keyword (https://metomi.github.io/fcm/doc/user_guide/code_management.html#svn_basic_keywords) already set up to test the accompanying source code. For the purposes of this tutorial we will create a new one.

Type the follow to create a temporary repository (you can safely delete it after finishing this tutorial):

```
mkdir -p ~/rose-tutorial
svnadmin create ~/rose-tutorial/spaceship_repos
(cd $(mktemp -d); mkdir -p trunk/src; svn import -m "" . file://$HOME/rose-
↪tutorial/spaceship_repos)
```

We then need to link the project name `SPACESHIP` with this project. Creating the file and directory if they do not exist add the following line to the file `$HOME/.metomi/fcm/keyword.cfg`:

```
location{primary}[spaceship] = file:///home/user/rose-tutorial/spaceship_repos
```

Make sure the path on the right-hand side matches the location you specified in the `svnadmin` command.

Now you can checkout a working copy of your repository by typing:

```
mkdir -p ~/rose-tutorial/spaceship_working_copy
cd ~/rose-tutorial/spaceship_working_copy
fcm checkout fcm:spaceship_tr .
```

Finally populate your working copy by running (answering `y` to the prompt):

```
rose tutorial rose-stem .
```

7.10.2 `spaceship_command.f90`

Our Fortran program is `spaceship_command.f90`, which reads in an initial position and spaceship mass from one namelist, and a series of commands to apply thrust in three-dimensional space. It then uses Newtonian

mechanics to calculate a final position.

You will find it in the `src` directory. Have a look at it and see what it does.

7.10.3 The `spaceship` app

Create a new Rose app called `spaceship`:

```
mkdir -p rose-stem/app/spaceship
```

Paste the following configuration into a `rose-app.conf` file within that directory:

```
[command]
default=spaceship_command.exe

[file:spaceship.NL]
source=namelist:spaceship

[file:command.NL]
source=namelist:command

[namelist:spaceship]
mass=2.0
position=0.0,0.0,0.0

[namelist:command]
thrust(1,:) =  1.0,  0.0, 0.0, 1.0,  0.0, -1.0, -1.0, 0.0, 0.0,  0.0
thrust(2,:) =  0.0, -2.0, 0.0, 1.0,  1.0,  0.5, -1.0, 1.5, 0.0, -1.0
thrust(3,:) =  0.0,  1.0, 0.0, 1.0, -1.0,  1.0, -1.5, 0.0, 0.0, -0.5
```

7.10.4 The `fcm-make` app

We now need to provide the instructions for `fcm_make` to build the Fortran executable.

Create a new app called `fcm_make_spaceship` with an empty `rose-app.conf` file.

Inside this app create a subdirectory called `file` and paste the following into the `fcm-make.cfg` file within that directory:

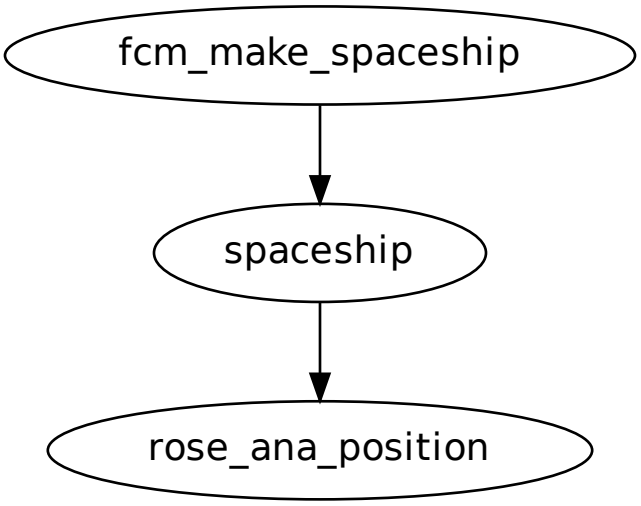
```
steps = build
build.source = $SOURCE_SPACESHIP/src
build.target{task} = link
```

The `$SOURCE_SPACESHIP` environment variable will be set using the Jinja2 variable of the same name which is provided by Rose Stem.

7.10.5 The `suite.rc` file

Next we will look at the `rose-stem/suite.rc` file.

The `suite.rc` file starts off with `UTC mode = True`, which you should already be familiar with. The next part is a Jinja2 block which links the group names the user can specify with the graph for that group. In this case, the group `command_spaceship` gives you the graph:



This variable `name_graphs` is used later to generate the graph when the suite is run. The Jinja2 variable `groups` is next. This enables you to set shortcuts to a list of groups, in this case specifying `all` on the command line will run the tasks associated with both `command_spaceship` and `fire_lasers`.

The scheduling section contains the Jinja2 code to use the information we have already set to generate the graph based on what the user requested on the command line.

The runtime section should be familiar. Note, however, that the `fcm_make_spaceship` task sets the environment variable `SOURCE_SPACESHIP` from the Jinja2 variable of the same name. This is how the variables passed with `--source` on the command line are passed to `fcm-make`, which then uses these environment variables in its own configuration files.

7.10.6 The `rose-suite.conf` file

The suites associated with Rose Stem require a version number indicating the version of the `rose stem` command with which they are compatible. This is specified in the `rose-suite.conf` file, together with the default values of `RUN_NAMES` and `SOURCE_SPACESHIP`. Paste the following into your `rose-suite.conf` file:

```
ROSE_STEM_VERSION=1

[jinja2:suite.rc]
RUN_NAMES=[]
SOURCE_SPACESHIP='fcm:spaceship_tr@head'
```

Both of the Jinja2 variables will be overridden by the user when they execute `rose stem` on the command line.

7.10.7 The `rose_ana_position` app

The final component is a `rose_ana` app to test whether the position of our spaceship matches the correct output.

Create an app named `rose_ana_position` and paste the following into its `rose-app.conf` file.

```
[ana:grepper.FilePattern(Check X position at each timestep)]
pattern='^\s*Position:\s*(.*?)\s*,'
files=/home/user/spaceship/kgo.txt
```

(continues on next page)

(continued from previous page)

```
     =../spaceship/output.txt

[ana:grepper.FilePattern(Check Y position at each timestep)]
pattern='^\s*Position:.*?,\s*(.*?)\s*,'
files=/home/user/spaceship/kgo.txt
     =../spaceship/output.txt

[ana:grepper.FilePattern(Check Z position at each timestep)]
pattern='^\s*Position:.*,\s*(.*)\s*$'
files=/home/user/spaceship/kgo.txt
     =../spaceship/output.txt
```

This will check that the positions reported by the program match those within the known good output file.

7.10.8 Known Good Output

In the root of the working copy is a file called `kgo.txt`.

The known good output should be the result of a control run. Rose Ana will compare the answers from this file (obtained using the extract and comparison methods in the `rose-app.conf` file) with the results from the user's code change.

Replace the `/home/user/spaceship` paths in the `rose_ana_position` app with the path to this file.

7.10.9 Adding the suite to version control

Before running the suite we need to make sure that all the files and directories we have created are known to the version control system.

Add all the new files you've created using `fcm add -c` *(answer yes to the prompts)*.

7.10.10 Running the test suite

We should now be able to run the test suite. Simply type:

```
rose stem --group=command_spaceship
```

anywhere in your working copy (the `--source` argument defaults to `.` so it should automatically pick up your working copy as the source).

Note: If your site uses a Cylc server, and your home directory is not shared with the Cylc server, you will need to add the option:

```
--host=localhost
```

We use `--group` in preference to `--task` in this suite (both are synonymous) as we specify a group of tasks set up in the Jinja2 variable `name_graphs`.

7.10.11 A failing test

Now edit the file:

```
rose-stem/app/spaceship/rose-app.conf
```

and change one of the thrusts, then rerun `rose stem`. You will find the `rose_ana_position` task fails, as the results have changed.

Try modifying the Fortran source code - for example, changing the direction in which thrust is applied (by changing the acceleration to be subtracted from the velocity rather than added). Again, rerun `rose stem`, and see the failure.

In this way, you can monitor whether the behaviour of code is changed by any of the code alterations you have made.

7.10.12 Further Exercises

If you wish, you can try extending the suite to include the `fire_lasers` group of tasks which was in the list of groups in the `suite.rc` file. Using the same technique as we've just demonstrated for piloting the spaceship, you should be able to aim and fire the ship's weapons.

7.10.13 Automatic Options

It is possible to automatically add options to `rose stem` using the `rose.conf[rose-stem]automatic-options` variable in the Site And User Configuration file. This takes the syntax of key-value pairs on a single line, and is functionally equivalent to adding them using the `-S` option on the `rose stem` command line. For example:

```
[rose-stem]
automatic-options=GRAVITY=newtonian PLANET=jupiter
```

sets the variable `GRAVITY` to have the value `newtonian`, and `PLANET` to be `jupiter`. These can then be used in the `suite.rc` file as Jinja2 variables.

7.11 Trigger

The `trigger` metadata item can be used to cut down the amount of irrelevant settings presented to the user in the command-rose-config-edit GUI by hiding any settings which are not relevant based on the value or state of other settings.

Irrelevant (`ignored` or `trigger-ignored`) settings do not get included in output files at runtime. In effect, they are commented out (`!` or `!!` prefix in Rose configurations).

7.11.1 Example

In this example, we'll be ordering pizza.

Create a new Rose application called `trigger`:

```
mkdir -p ~/rose-tutorial/trigger
cd ~/rose-tutorial/trigger
```

Create a `rose-app.conf` file that looks like this:

```
[command]
default=order.exe

[env]
BUDGET=10

[file:order.nl]
source=namelist:pizza_order namelist:side_order
```

(continues on next page)

(continued from previous page)

```
[namelist:pizza_order]
extra_chicken=.false.
pepperoni_multiple=1
no_mushrooms=.false.
pizza_type='Veggie Supreme'
truffle='none'

[namelist:side_order]
garlic_bread=.false.
soft_drink=.false.
```

We'll add some metadata to make it nice. Create a `meta/` sub-directory with a `rose-meta.conf` file that looks like this:

```
[env]

[env=BUDGET]
type=integer

[file:order.nl]

[namelist:pizza_order]

[namelist:pizza_order=extra_chicken]
type=logical

[namelist:pizza_order=pepperoni_multiple]
values=1,2,3

[namelist:pizza_order=no_mushrooms]
type=logical

[namelist:pizza_order=pizza_type]
sort-key=00-type
values='Veggie Supreme', 'Pepperoni', 'BBQ Chicken'

[namelist:pizza_order=truffle]
values='none', 'white', 'black'

[namelist:side_order]

[namelist:side_order=garlic_bread]
type=logical

[namelist:side_order=soft_drink]
type=logical
```

Once you've done that, run command-rose-config-edit in the application directory and navigate around the pages.

There are quite a lot of settings that are only relevant in certain contexts - for example, `namelist:pizza_order=extra_chicken` is pretty irrelevant if we're ordering a `'Veggie Supreme'`.

7.11.2 Adding Triggers

Let's add some trigger information.

In the `rose-meta.conf` file, under `[namelist:pizza_order=pizza_type]`, add:

```
trigger=namelist:pizza_order=extra_chicken: 'BBQ Chicken';
        namelist:pizza_order=pepperoni_multiple: 'Pepperoni', 'BBQ Chicken';
```

This states which values of `pizza_type` are relevant for which settings. This means that `extra_chicken` is only relevant when `pizza_type` is `'BBQ Chicken'` - otherwise, it should be in an ignored state. `pepperoni_multiple` is relevant for more than one value of `pizza_type`.

We should also make sure we don't order over our budget, especially by splashing out on truffles. Add the following to `[env=BUDGET]`:

```
trigger=namelist:pizza_order=truffle: this > 25;
        namelist:side_order: this >= 10;
```

See app-meta-mini-lang for details on this syntax.

What we've done here is use a small subset of the Rose configuration metadata logical syntax to specify a range of allowed values (the `this > 25` part). Here, `this` is a placeholder for the value of `env=BUDGET`; the expression syntax is essentially Pythonic.

We've also specified a section `namelist:side_order` in the trigger, which is perfectly valid - this means that the whole section and its options will be ignored when the value of `env=BUDGET` is below 10. The truffle option will be ignored unless `env=BUDGET` is more than 25.

7.11.3 Fixing Trigger Errors

If we load the config editor (or reload the metadata) again, we should get some trigger errors. These essentially say that some of our settings are in the wrong state now - in our case, they should be `trigger-ignored`.

You can fix them on the command line by running `rose macro --fix` or `rose macro -F` in the app directory (one level up from the meta directory) - this is what you would do if you were working with a text editor and made changes to values.

Similarly, you can run "Autofix" in the config editor. You can do this in three ways:

- By clicking the *Metadata → Autofix all configurations* menu.
- Using the *Auto-fix* toolbar button.
- Or via the right-click menu for the root page in the left-hand tree panel, in this case `pizza_order`.

Run "Autofix" in one of the above ways.

Results

If you accept the changes, the state of these settings will be corrected - if you go to the page, you'll see that they've vanished! They're actually just commented out, and viewable via the menu *View → View All Ignored Variables*.

Try altering the values of `namelist:pizza_order=pizza_type` and `env=BUDGET` with *View → View All Ignored Variables* on and off. This should enable and `trigger-ignore` different settings.

When `env=BUDGET` is below 10, the `namelist:side_order` section will be `trigger-ignored`, and the `garlic_bread` and `soft_drink` will be `section-ignored` - ignored because their parent section is ignored.

You can get more information about why an option is ignored in the config editor by hovering over its ignored flag, or looking at the option's menu button `Info` entry.

Setting ids mentioned in the `Info` dialog are usually clickable links, so you can go directly to the relevant id.

7.11.4 Multiple Inheritance

More than one setting can decide whether something is relevant. In that case, the subject is relevant only if all the parents agree that it is - an AND relationship.

For example, we already have one trigger for `namelist:pizza_order=truffle` (`env=BUDGET`) - but it should also only be relevant when `namelist:pizza_order=no_mushrooms` is `.false.`.

Open the metadata file in a text editor, and add the following to the `[namelist:pizza_order=no_mushrooms]` metadata section:

```
trigger=namelist:pizza_order=truffle: .false.
```

This means that the `namelist:pizza_order=truffle` option will only be enabled when `env=BUDGET` is greater than 25 (our older trigger) and `namelist:pizza_order=no_mushrooms` is `.false.`.

Save the metadata file and reload the metadata in the config editor, and test it for yourself.

7.11.5 Cascading Triggering

Triggering is not just based on values - if a setting is missing or `trigger-ignored`, any settings that it triggers will be `trigger-ignored` by default i.e. triggers can act in a cascade - A triggers B triggers C.

We can see this by replacing the `env=BUDGET` trigger with:

```
trigger=namelist:pizza_order=truffle: this > 25;
        namelist:side_order: this >= 10;
        namelist:pizza_order=pizza_type: this >= 5;
```

When `env=BUDGET` is less than 5, `namelist:pizza_order=pizza_type` will be `trigger-ignored`. This means that all of its triggered settings like `namelist:pizza_order=extra_chicken` are irrelevant and will also be `trigger-ignored`.

We need to add `no_mushrooms` to the `[namelist:pizza_order=pizza_type]` section so that it is `trigger-ignored` when no pizza can be ordered - replace the `[namelist:pizza_order=pizza_type]` trigger with:

```
trigger=namelist:pizza_order=extra_chicken: 'BBQ Chicken';
        namelist:pizza_order=pepperoni_multiple: 'Pepperoni', 'BBQ Chicken';
        namelist:pizza_order=no_mushrooms;
```

Save, reload, and try changing `env=BUDGET` below 5 to see what it does to the options in `namelist:pizza_order`.

7.11.6 Triggering Based On State

There's also another way to express a trigger - you don't have to express a value or range of values in a trigger expression.

Quite often you only want a setting to be `trigger-ignored` or enabled purely based on the availability of another setting - whether it is present and whether it is `trigger-ignored`. You might not care what particular value it has.

This can be expressed by adding a trigger but omitting the value part of the syntax. Let's add an option that we can use.

Add a new variable in the metadata by adding these lines to the metadata file:

```
[namelist:pizza_order=dip_type]
values='Garlic','Sour Cream','Salsa','Brown Sauce','Mustard'
```

We should add a trigger expression as well - replace the `[namelist:pizza_order=pizza_type]` trigger with:

```
trigger=namelist:pizza_order=extra_chicken: 'BBQ Chicken';
        namelist:pizza_order=pepperoni_multiple: 'Pepperoni', 'BBQ Chicken';
        namelist:pizza_order=no_mushrooms;
        namelist:pizza_order=dip_type;
```

This means that `namelist:pizza_order=dip_type` is dependent on `namelist:pizza_order=pizza_type`, and will only be ignored when that is ignored - but the value of `pizza_type` doesn't matter to it.

Save the file and reload the metadata in the config editor. We'll need to add the `namelist:pizza_order=dip_type` to use it properly - you can do this from the `namelist:pizza_order` page via:

- The *Add* toolbar button.
- The right-click page menu.
- The *View* → *View Latent Variables* menu.

After enabling the view, you should see `dip_type` appear as an option that could be added. It will already have the correct triggered state (the same state as `namelist:pizza_order=pizza_type`) - verify for yourself that this works! You can then just add it via the menu button for the option.

7.11.7 Further Reading

For more information see Metadata.

7.12 Upgrading

As apps are developed, newer metadata versions can be created each time the application inputs are changed, or just between major releases.

This may mean, for example, that a new compulsory option is added or an old one is removed.

Upgrade macros may be written to automatically apply these changes.

Upgrade macros are used to upgrade Rose apps to newer metadata versions. They are intended to keep application configurations in sync with changes to application inputs e.g. from new code releases.

This part tutorial walks you through upgrading applications.

7.12.1 Example

Create a new Rose application called `garden`:

```
mkdir -p ~/rose-tutorial/garden
cd ~/rose-tutorial/garden
```

Create within it a `rose-app.conf` file that looks like this:

```
meta=rose-demo-upgrade/garden0.1

[env]
FOREST=true

[namelist:features]
rose_bushes=2
```

The `meta=...` line references a category (`rose-demo-upgrade`) at a particular version (`garden0.1`). It's the version that we want to change.

7.12.2 rose app-upgrade

Change directory to your new application directory. You can see the available upgrade versions for your new app config by running:

```
rose app-upgrade
```

This gives you a list of versions to upgrade to - see the help for more information (run `rose help app-upgrade`).

There can often be more versions than you can see by just running command-rose-app-upgrade. They will not have formal metadata, and represent intermediary steps along the way between proper named versions. You can see all the possible versions by running:

```
rose app-upgrade --all-versions
```

You can upgrade directly to the latest (`garden0.9`) or to other versions - let's choose `garden0.2` to start with. Run:

```
rose app-upgrade garden0.2
```

7.12.3 Upgrade Changes

This will give you a list of changes that the upgrade will apply to your configuration. Accept it, and your application configuration will be upgraded, with a new option (`shrubberies`) and a new `meta=...` version of the metadata to point to. Have a look at the changed `rose-app.conf` if you like.

Try repeating this by upgrading to `garden0.3` in the same way. This time, you'll get a warning - warnings are used to point out problems such as deprecated options when you upgrade.

We can upgrade over many versions at once - for example, directly to `garden0.9` - and the changes between each version will be aggregated into a single list of changes.

Try running:

```
rose app-upgrade garden0.9
```

If you accept the changes, your app config will be upgraded through all the intermediary versions to the new one. Have a look at the `rose-app.conf` file.

If you run Rose command-rose-app-upgrade with no arguments, you can see that you're using the latest version.

7.12.4 Downgrading

Some versions may support downgrading - the reverse operation to upgrading. You can see if this is supported by running:

```
rose app-upgrade --downgrade
```

You can then use it to downgrade by running:

```
rose app-upgrade --downgrade <VERSION>
```

where `VERSION` is a lower supported version. This time, some settings may be removed.

Tip: See also:

- conf-meta
- rose-upgr-macros

7.13 Upgrading Macro Development

Upgrade macros are used to upgrade Rose apps to newer metadata versions. They are intended to keep application configurations in sync with changes to application inputs e.g. from new code releases.

You should already be familiar with using command-rose-app-upgrade (see the *Upgrading tutorial* (page 76) and the concepts in the reference material).

7.13.1 Example

In this example, we'll be upgrading a boat on a desert island.

Create a Rose application called `make-boat-app`:

```
mkdir -p ~/rose-tutorial/make-boat-app
cd ~/rose-tutorial/make-boat-app
```

Create a `rose-app.conf` file with the following content:

```
meta=make-boat/0.1

[namelist:materials]
hollow_tree_trunks=1
paddling_twigs=1
```

You now have a Rose application configuration that configures our simple boat (a dugout canoe). It references a meta flag (for which metadata is unlikely to already exist), made up of a category (`make-boat`) at a particular version (`0.1`). The meta flag is used by Rose to locate a configuration metadata directory.

Make sure you're using `make-boat` and not `make_boat` - the hyphen makes all the difference!

Note: The version in the meta flag doesn't have to be numeric - it could be `vn0.1` or `alpha` or `Crafty-Canoe`.

We need to create some metadata to make this work.

7.13.2 Example Metadata

We need a `rose-meta/` directory somewhere, to store our metadata - for the purposes of this tutorial it's easiest to put in in your homespace, but the location does not matter.

Create a `rose-meta/make-boat/` directory in your homespace:

```
mkdir -p ~/rose-meta/make-boat/
```

This is the category (also called command) directory for the metadata, which will hold sub-directories for actual configuration metadata versions (each containing a `rose-meta.conf` file, etc).

N.B. Configuration metadata would normally be managed by whoever manages Rose installation at your site.

We know we need some metadata for the `0.1` version, so create a `0.1/` subdirectory under `rose-meta/make-boat/`:

```
mkdir ~/rose-meta/make-boat/0.1/
```

We'll need a `rose-meta.conf` file there too, so create an empty one in the new directory:

```
touch ~/rose-meta/make-boat/0.1/rose-meta.conf
```

We can safely say that our two namelist inputs are essential for the construction and testing of the boat, so we can paste the following into the newly created `rose-meta.conf` file:

```
[namelist:materials=hollow_tree_trunks]
compulsory=true
values=1

[namelist:materials=paddling_twigs]
compulsory=true
range=1:
type=integer
```

So far, we have a normal application configuration which references some metadata, somewhere, for a category at a certain version.

Let's make another version to upgrade to.

The next version of our boat will have outriggers (https://en.wikipedia.org/wiki/Outrigger_canoe) to make it more stable. Some of the inputs in our application configuration will need to change.

Our application configuration might need to look something like this, after any upgrade (don't change it yet!):

```
meta=make-boat/0.2

[namelist:materials]
hollow_tree_trunks=1
misc_branches=4
outrigger_tree_trunks=2
paddling_branches=1
```

It looks like we've added the inputs `misc_branches`, `outrigger_tree_trunks` and `paddling_branches`. `paddling_twigs` is now no longer there (now redundant), so we can remove it from the configuration when we upgrade.

Let's create the new metadata version, to document what we need and don't need.

Create a new subdirectory under `make-boat/` called `0.2/` containing a `rose-meta.conf` file that looks like this:

```
[namelist:materials=hollow_tree_trunks]
compulsory=true
values=1

[namelist:materials=misc_branches]
compulsory=true
range=4:

[namelist:materials=paddling_branches]
compulsory=true
range=1:
type=integer

[namelist:materials=outrigger_tree_trunks]
```

(continues on next page)

(continued from previous page)

```
compulsory=true
values=2
```

You can check that everything is OK so far by changing directory to the `make-boat/` directory and running `find` - it should look something like:

```
.
./0.1
./0.1/rose-meta.conf
./0.2
./0.2/rose-meta.conf
```

We now want to automate the process of updating our app config from `make-boat/0.1` to the new `make-boat/0.2` version.

7.13.3 `versions.py`

Upgrade macros are invoked through a Python module, `versions.py`, that doesn't live with any particular version metadata - it should be present at the root of the category directory.

Create a new file `versions.py` under `make-boat/` (`~/rose-meta/make-boat/versions.py`). We'll add a macro to it in a little bit.

Upgrade Macros Explained

Upgrade macros are Python objects with a `BEFORE_TAG` (e.g. `"0.1"`) and an `AFTER_TAG` (e.g. `"0.2"`). The `BEFORE_TAG` is the 'start' version (if upgrading) and the `AFTER_TAG` is the 'destination' version.

When a user requests an upgrade for their configuration (e.g. by running command-rose-app-upgrade), the `versions.py` file will be searched for a macro whose `BEFORE_TAG` matches the `meta=...` version.

For example, for our `meta=make-boat/0.1` flag, we'd need a macro whose `BEFORE_TAG` was `"0.1"`.

When a particular upgrade macro is run, the version in the app configuration will be changed from `BEFORE_TAG` to `AFTER_TAG` (e.g. `meta=make-boat/0.1` to `meta=make-boat/0.2`), as well as making other changes to the configuration if needed, like adding/removing the right variables.

If the user wanted to upgrade across multiple versions - e.g. `0.1` to `0.4` - there would need to be a chain of objects whose `BEFORE_TAG` was equal to the last `AFTER_TAG`, ending in an `AFTER_TAG` of `0.4`.

We'll cover multiple version upgrading later in the tutorial.

Upgrade Macro Skeleton

Upgrade macros are bits of Python code that essentially look like this:

```
class Upgrade272to273(rose.upgrade.MacroUpgrade):

    """Upgrade from 27.2 to 27.3."""

    BEFORE_TAG = "27.2"
    AFTER_TAG = "27.3"

    def upgrade(self, config, meta_config=None):
        """Upgrade the application configuration (config)."""
        # Some code doing something to config goes here.
        return config, self.reports
```

They are sub-classes of a particular class, `rose.upgrade.MacroUpgrade`, which means that some of the Python functionality is done 'under the hood' to make things easier.

You shouldn't need to know very much Python to get most things done.

Example Upgrade Macro

Paste the following into your `versions.py` file:

```
import rose.upgrade


class MyFirstUpgradeMacro(rose.upgrade.MacroUpgrade):

    """Upgrade from 0.1 (Canonical Canoe) to 0.2 (Outrageous Outrigger)."""

    BEFORE_TAG = "0.1"
    AFTER_TAG = "0.2"

    def upgrade(self, config, meta_config=None):
        """Upgrade the boat!"""
        # Some code doing something to config goes here.
        return config, self.reports
```

This is already a functional upgrade macro - although it won't do anything.

Note: The name of the class (`MyFirstUpgradeMacro`) doesn't need to be related to the versions - the only identifiers that matter are the `BEFORE_TAG` and the `AFTER_TAG`.

We need to get the macro to do the following:

- add the option `namelist:materials=misc_branches`
- add the option `namelist:materials=outrigger_tree_trunks`
- add the option `namelist:materials=paddling_branches`
- remove the option `namelist:materials=paddling_twigs`

We can use the rose-upgr-macros provided to express this in Python code. Replace the `# Some code doing something...` line with:

```
self.add_setting(config, ["namelist:materials", "misc_branches"], "4")
self.add_setting(
        config, ["namelist:materials", "outrigger_tree_trunks"], "2")
self.add_setting(
        config, ["namelist:materials", "paddling_branches"], "1")
self.remove_setting(config, ["namelist:materials", "paddling_twigs"])
```

This changes the app configuration (`config`) in the way we want, and (behind the scenes) adds some things to the `self.reports` list mentioned in the `return config, self.reports` line.

Note: When we add options like `misc_branches`, we must specify default values to assign to them.

Tip: Values should always be specified as strings e.g. (`"1"` rather than `1`).

Customising the Output

The methods `self.add_setting` and `self.remove_setting` will provide a default message to the user about the change (e.g. `"Added X with value Y"`), but you can customise them to add your own using the info 'keyword argument' like this:

```
self.add_setting(
    config, ["namelist:materials", "outrigger_tree_trunks"], "2",
    info="This makes it into a trimaran!")
```

If you want to, try adding your own messages.

Running `rose app-upgrade`

Our upgrade macro will now work - change directory to the application directory and run:

```
rose app-upgrade --meta-path=~/rose-meta/
```

This should display some information about the current and available versions - see the help by running `rose help app-upgrade`.

`--meta-path` equals the path to the `rose-meta/` directory you created - as this path isn't configured in the site/user configuration, we need to set it manually. This won't normally be the case for users, if the metadata is centrally managed.

Let's upgrade to `0.2`. Run:

```
rose app-upgrade --meta-path=~/rose-meta/ 0.2
```

This should provide you with a summary of changes (including any custom messages you may have added) and prompt you to accept them. Accept them and have a look at the app config file - it should have been changed accordingly.

Using Patch Configurations

For relatively straightforward changes like the one above, we can configure a macro to apply patches to the configuration without having to write setting-specific Python code.

We'll add a rudder option for our `0.3` version, with a `namelist:materials=l_rudder_branch`.

Create a `0.3` directory in the same way that you created the `0.1` and `0.2` metadata directories. Add a `rose-meta.conf` file that looks like this:

```
[namelist:materials=hollow_tree_trunks]
compulsory=true
values=1

[namelist:materials=l_rudder_branch]
compulsory=true
type=logical

[namelist:materials=misc_branches]
compulsory=true
type=integer
range=4:

[namelist:materials=outrigger_tree_trunks]
compulsory=true
values=2

[namelist:materials=paddling_branches]
```

(continues on next page)

(continued from previous page)

```
compulsory=true
range=1:
type=integer
```

We need to write another macro in `versions.py` - append the following code:

```
class MySecondUpgradeMacro(rose.upgrade.MacroUpgrade):

    """Upgrade from 0.2 (Outrageous Outrigger) to 0.3 (Amazing Ama)."""

    BEFORE_TAG = "0.2"
    AFTER_TAG = "0.3"

    def upgrade(self, config, meta_config=None):
        """Upgrade the boat!"""
        self.act_from_files(config)
        return config, self.reports
```

The `self.act_from_files` line tells the macro to look for patch configuration files - two files called `rose-macro-add.conf` and `rose-macro-remove.conf`, under an `etc/BEFORE_TAG/` subdirectory - in our case, `~/rose-meta/make-boat/etc/0.2/`.

Whatever is found in `rose-macro-add.conf` will be added to the configuration, and whatever is found in `rose-macro-remove.conf` will be removed. If the files don't exist, nothing will happen.

Let's configure what we want to happen. Create a directory `~/rose-meta/make-boat/etc/0.2/`, containing a `rose-macro-add.conf` file that looks like this:

```
[namelist:materials]
l_rudder_branch=.true.
```

Note: If a `rose-macro-add.conf` setting is already defined, the value of `l_rudder_branch` will not be overwritten. In our case, we don't need a `rose-macro-remove.conf` file.

Go ahead and upgrade the app configuration to `0.3`, as you did before.

The `rose-app.conf` should now contain the new option, `l_rudder_branch`.

More Complex Upgrade Macros

The rose-upgr-macros gives us quite a bit of power without having to write too much Python.

For our `1.0` release we want to make some improvements to out sailing equipment:

- We want to increase the number of `misc_branches` to be at least 6.
- We want to add a `sail_canvas_sq_m` option.

We may want to issue a warning for a deprecated option (`paddle_branches`) so that the user can decide whether to remove it.

Create the file `~/rose-meta/make-boat/1.0/rose-meta.conf` and paste in the following configuration:

```
[namelist:materials=hollow_tree_trunks]
compulsory=true
values=1

[namelist:materials=l_rudder_branch]
compulsory=true
```

(continues on next page)

(continued from previous page)

```
type=logical

[namelist:materials=misc_branches]
compulsory=true
range=6:
type=integer

[namelist:materials=outrigger_tree_trunks]
compulsory=true
values=2

[namelist:materials=paddling_branches]
range=0:
type=integer
warn-if=True # Deprecated - real sailors don't use engines

[namelist:materials=sail_canvas_sq_m]
range=4:
type=real
```

We need to write a macro that reflects these changes.

We need to start with appending the following code to `versions.py`:

```
class MyMoreComplexUpgradeMacro(rose.upgrade.MacroUpgrade):

    """Upgrade from 0.3 (Amazing Ama) to 1.0 (Tremendous Trimaran)."""

    BEFORE_TAG = "0.3"
    AFTER_TAG = "1.0"

    def upgrade(self, config, meta_config=None):
        """Upgrade the boat!"""
        # Some code doing something to config goes here.
        return config, self.reports
```

We already know how to add an option, so replace `# Some code going here...` with `self.add_setting(config, ["namelist:materials", "sail_canvas_sq_m"], "5")`

To perform the check/change in the number of `misc_branches`, we can insert the following lines after the one we just added:

```
branch_num = self.get_setting_value(
       config, ["namelist:materials", "misc_branches"])
if branch_num.isdigit() and float(branch_num) < 6:
    self.change_setting_value(
             config, ["namelist:materials", "misc_branches"], "6")
```

This extracts the value of `misc_branches` (as a string!) and if the value represents a positive integer that is less than 6, changes it to `"6"`. It's good practice to guard against the possibility that a user might have set the value to a non-integer representation like `'many'` - if we don't do this, the macro may crash out when running things like `float`.

In a similar way, to flag a warning, insert:

```
paddles = self.get_setting_value(
               config, ["namelist:materials", "paddling_branches"])
if paddles is not None:
    self.add_report("namelist:materials", "paddling_branches",
                    paddles, info="Deprecated - probably not needed.",
                    is_warning=True)
```

This calls `self.add_report` if the option `paddling_branches` is present. This is a method that notifies the user of actions and issues by appending things to the `self.reports` list which appears on the `return ...` line.

Run `rose app-upgrade --meta-path=~/rose-meta/ 1.0` to see the effect of your changes. You should see a warning message for `namelist:materials=paddling_branches` as well.

Upgrading Many Versions at Once

We've kept in step with the metadata by upgrading incrementally, but typically users will need to upgrade across multiple versions. When this happens, the relevant macros will be applied in turn, and their changes and issues aggregated.

Turn back the clock by reverting your application configuration to look like it was at `0.1`:

```
meta=make-boat/0.1

[namelist:materials]
hollow_tree_trunks=1
paddling_twigs=1
```

Run `rose app-upgrade --meta-path=~/rose-meta/` in the application directory. You should see that the version has been downgraded to 0.1, the available versions to upgrade to should also be listed - let's choose `1.0`. Run:

```
rose app-upgrade --meta-path=~/rose-meta/ 1.0
```

This should aggregate all the changes that our macros make - if you accept the changes, it will upgrade all the way to the `1.0` version we had before.

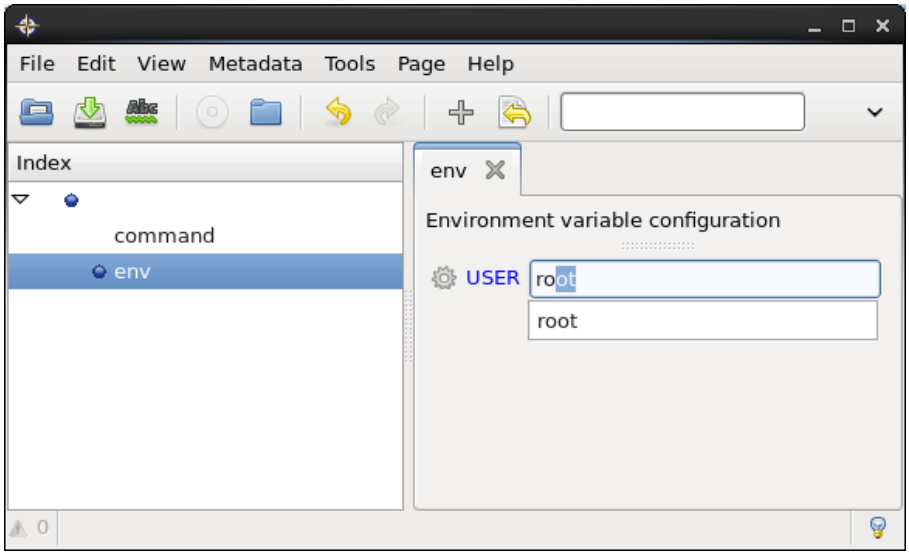
Tip: See also:

- rose-upgr-macros
- api-rose-macro

7.14 Widget Development

The command-rose-config-edit GUI displays configurations using built-in widgets. For more complex requirements command-rose-config-edit supports custom widgets as plugins.

In this tutorial we will write a custom widget which offers typing suggestions when entering usernames.



Warning: If you find yourself needing to write a custom widget, please contact the Rose team for guidance.

7.14.1 Example

Create a new Rose app by running the following command replacing `DIRECTORY` with the path in which to create the suite:

```
rose tutorial widget <DIRECTORY>
cd <DIRECTORY>
```

You will now have a Rose app which contains the following files:

```
<DIRECTORY>/
|-- meta/
|   `-- lib/
|       `-- python/
|           `-- widget/
|               |-- __init__.py
|               `-- username.py
`-- rose-app.conf
```

The `rose-app.conf` file defines an environment variable called `USER`:

```
[env]
USER=fred
```

The `__init__.py` file is empty - the presence of this file declares the `widget` directory as a python package (https://docs.python.org/3/tutorial/modules.html#packages).

The `username.py` file is where we will write our widget.

Initial Code

We will start with a slimmed-down copy of the class `rose.config_editor.valuewidget.text.RawValueWidget` which you will find in the file `username.py`. It contains all the API calls you would normally ever need.

We are now going to extend the widget to be more useful.

Add a line importing the `pwd` package at the top of the file:

```
+ import pwd

 import gobject
 import pygtk
 pygtk.require('2.0')
 import gtk
```

This adds the Python library that we'll use in a minute.

Now we need to create a predictive text model by adding some data to our `gtk.Entry` text widget.

We need to write our method `_set_completion`, and put it in the main body of the class. This will retrieve usernames from the `pwd.getpwall()` function and store them so they can be used by the text widget `self.entry`.

Add the following method to the `UsernameValueWidget` class:

```
def _set_completion(self):
    # Return a predictive text model.
    completion = gtk.EntryCompletion()
    model = gtk.ListStore(str)
    for username in [p.pw_name for p in pwd.getpwall()]:
        model.append([username])
    completion.set_model(model)
    completion.set_text_column(0)
    completion.set_inline_completion(True)
    self.entry.set_completion(completion)
```

We need to make sure this method gets called at the right time, so we add the following line to the `__init__` method:

```
  self.entry.show()
+ gobject.idle_add(self._set_completion)
  self.pack_start(self.entry, expand=True, fill=True,
                  padding=0)
```

We could just call `self._set_completion()` there, but this would hang the config editor while the database is retrieved.

Instead, we've told GTK to fetch the predictive text model when it's next idle (`gobject.idle_add`). This means it will be run after it finishes loading the page, and will be more-or-less invisible to the user. This is a better way to launch something that may take a second or two. If it took any longer, we'd probably want to use a separate process.

Referencing the Widget

Now we need to refer to it in the metadata to make use of it.

Create the file `meta/rose-meta.conf` and paste the following configuration into it:

```
[env=USER]
widget[rose-config-edit]=username.UsernameValueWidget
```

This means that we've set our widget up for the option `USER` under the section *env*. It will now be used as the widget for this variable's value.

Results

Try opening up the config editor in the application directory (where the `rose-app.conf` is) by running:

```
rose config-edit
```

Navigate to the *env* page. You should see your widget on the page! As you type, it should provide helpful auto-completion of usernames. Try typing your own username.

7.14.2 Further Reading

For more information, see api-gtk and the PyGTK (http://www.pygtk.org/) web page.

/(str:prefix)

```
GET (str:prefix)/get_known_keys, ??
GET (str:prefix)/get_optional_keys, ??
GET (str:prefix)/get_query_operators,
      ??
GET (str:prefix)/query, ??
GET (str:prefix)/search, ??
```

INDEX