



# Rose Documentation

*Release 2018.06.0*

**Metomi**

**Jun 27, 2018**



# USER GUIDE

<b>1 Installation</b>	<b>3</b>
<b>2 Getting Started</b>	<b>11</b>
<b>3 Cylc Tutorial</b>	<b>13</b>
<b>4 Rose Tutorial</b>	<b>97</b>
<b>5 Cheat Sheet</b>	<b>175</b>
<b>6 Glossary</b>	<b>179</b>
<b>7 Rose Configuration</b>	<b>193</b>
<b>8 Command Reference</b>	<b>245</b>
<b>9 Rose Environment Variables</b>	<b>275</b>
<b>10 Rose Bash Library</b>	<b>281</b>
<b>11 Rose Built-In Applications</b>	<b>283</b>
<b>12 Rose GTK library</b>	<b>295</b>
<b>13 Rose Macro API</b>	<b>303</b>
<b>14 Rose Upgrade Macro API</b>	<b>309</b>
<b>15 Rosie Web API</b>	<b>313</b>
<b>16 Other</b>	<b>317</b>
<b>HTTP Routing Table</b>	<b>319</b>
<b>Python Module Index</b>	<b>321</b>
<b>Index</b>	<b>323</b>



Rose is a toolkit for writing, editing and running application configurations. *What Is Rose* (page 97)?



Rose uses the [Cylc](http://cylc.github.io/cylc/) (<http://cylc.github.io/cylc/>) workflow engine for running suites of inter-dependent applications.

*What Is Cylc?* (page 13)



(<http://cylc.github.io/cylc/>)



## INSTALLATION

The source code for Rose is available via [GitHub](https://github.com/metomi/rose) (<https://github.com/metomi/rose>), code releases are available in `zip` and `tar.gz` [archives](https://github.com/metomi/rose/tags) (<https://github.com/metomi/rose/tags>).

1. Un-pack the archive file into an appropriate location on your system.
2. Add the `rose/bin/` directory into your PATH environment variable.
3. Check system compatibility by running `rose check-software` (page 247).

### 1.1 System Requirements

Rose runs on Unix/Linux systems and is known to run on RHEL6 and a number of systems including those documented under [metomi-vms](https://github.com/metomi/metomi-vms) (<https://github.com/metomi/metomi-vms>).

System compatibility can be tested using the `rose check-software` (page 247) command.

#### Required Software

- python - 2.6.6+, <3
- cylc
- py:pygtk - 2.12+
- py:jinja2

#### Rose Bush

- py:cherrypy

#### Rosie

- py:cherrypy
- py:requests - 2.2.1+
- py:sqlalchemy - 0.6.9+
- svn - 1.8+
- fcm
- perl - 5.10.1+

#### Documentation Builder

- python - 2.7+, <3
- graphviz-dot
- tex
- pdflatex
- py:sphinx - 1.5.3+

- py:sphinx\_rtd\_theme - 0.2.4+
- py:sphinxcontrib.httpdomain
- py:hieroglyph

#### Documentation Builder - PDF Dependencies

- latexmk

#### Documentation Builder - Linkcheck Dependencies

- py:requests

## 1.2 Host Requirements

Whilst you can install and use Rose & Cylc on a single host (i.e. machine), most installations are likely to be more complex. There are five types of installation:

### 1.2.1 Hosts For Running Cylc Suites

Each user can just run their suites on the host where they issue the *rose suite-run* (page 263) command. However, the local host may not meet the necessary requirements for connectivity (to the hosts running the tasks) or for availability (the host needs to remain up for the duration of the suite). Therefore, Rose can be configured to run the suites on remote hosts. If multiple hosts are available, by default the host with the lowest system load average will be used.

#### Installation requirements:

- Rose, Cylc, Bash, Python, jinja2.
- Subversion & FCM (*only if you want rose suite-run* (page 263) to install files from Subversion using FCM keywords).

#### Connectivity requirements:

- Must be able to submit tasks to the hosts which run the suite tasks, either directly or via SSH access to another host.

### 1.2.2 Hosts For Running Tasks In Suites

#### Installation requirements:

- Rose, Cylc, Bash, Python.
- Subversion & FCM *only if you want to use the Rose install utility to install files from Subversion using FCM keywords.*

#### Connectivity requirements:

- Must be able to communicate back to the hosts running the Cylc suites via HTTPS (preferred), HTTP or SSH.

### 1.2.3 Hosts For Running User Interactive Tools

#### Installation requirements:

- Rose, Cylc, Bash, Python, requests, Subversion, FCM, Pygraphviz (+ graphviz), PyGTK (+ GTK).

#### Connectivity requirements:

- Must have HTTP access to the hosts running the Rosie web service.

- Must have access to the Rosie Subversion repositories via the appropriate protocol.
- Must have HTTP and SSH access to the hosts running the Cylc suites.
- Must share user accounts and \$HOME directories with the hosts running the Cylc suites.

### 1.2.4 Hosts For Running Rose Bush

**Installation requirements:**

- Rose, Bash, Python, cherrypy, jinja2.

**Connectivity requirements:**

- Must be able to access the home directories of users' Cylc run directories.

### 1.2.5 Hosts For Rosie Subversion Repositories And The Rosie Web Services

Typically you will only need a single host but you can have multiple repositories on different hosts if you require this.

**Installation requirements:**

- Rose, Bash, Python, cherrypy, jinja2, sqlalchemy, Subversion.

---

**Note:** This section has assumed that you wish to use Rose & Cylc (including their respective GUIs) and use Rosie to store your suites. However, there are many ways of using Rose. For instance:

- You can use Rose without using cylc.
  - You can use Rose & Cylc but not use Rosie.
  - You can use Rose & Cylc from the command line without using their GUIs.
- 

## 1.3 Configuring Rose

**lib/bash/rose\_init\_site** If you do not have root access to install the required Python libraries, you may need to edit this file to ensure Python libraries are included in the PYTHONPATH environment variable. For example, if you have installed some essential Python libraries in your HOME directory, you can add a lib/bash/rose\_init\_site file with the following contents:

```
# Essential Python libraries installed under
# "/home/daisy/usr/lib/python2.6/site-packages/"
if [[ -n "${PYTHONPATH:-}" ]]; then
    PYTHONPATH="/home/daisy/usr/lib/python2.6/site-packages:${PYTHONPATH}"
else
    PYTHONPATH="/home/daisy/usr/lib/python2.6/site-packages"
fi
export PYTHONPATH
```

**etc/rose.conf** You should add/edit this file to meet the requirement of your site. Examples can be found at the etc/rose.conf.example file in your Rose distribution. See also *rose.conf* (page 198) in the API reference.

## 1.4 Configuring Rosie Client

Rosie is an optional suite storage and discovery system.

Rosie stores suites using Subversion repositories, with databases behind a web interface for suite discovery and lookup.

If users at your site are able to access Rosie services on the Internet or if someone else has already configured Rosie services at your site, all you need to do is configure the client to talk to the servers. Refer to the [Configuring a Rosie Server](#) (page 8) section if you need to configure a Rosie server for your site.

To set up the Rosie client for the site, add/modify the `rose.conf[rosie-id]` (page 204) E.g.:

```
[rosie-id]
prefix-default=x
prefixes-ws-default=x myorg

prefix-location.x=https://somehost.on.the.internet/svn/roses-x
prefix-web.x=https://somehost.on.the.internet/trac/roses-x/intertrac/source:
prefix-ws.x=https://somehost.on.the.internet/rozie/x

prefix-location.myorg=svn://myhost.myorg/roses-myorg
prefix-web.myorg=http://myhost.myorg/trac/roses-myorg/intertrac/source:
prefix-ws.myorg=http://myhost.myorg/rozie/myorg
```

Check the following:

1. You can access the Rosie Subversion repository without being prompted for a username and a password. This may require configuring Subversion to cache your authentication information with a keyring.  
*(See Subversion Book > Advanced Topics > Network Model > Client Credentials for a discussion on how to do this.)*
2. The Rosie web service is up and running and you can access the Rosie web service from your computer. E.g. if the Rosie web service is hosted at `https://somehost.on.the.internet/rozie/x`, you can check that you have access by typing the following on the command line:

```
curl -I https://somehost.on.the.internet/rozie/x
```

It should return a HTTP code 200. If you are prompted for a username and a password, you may need to have access to a keyring to cache the authentication information.

3. You can access the Rosie web service using the Rosie client. E.g. using the above configuration for the prefix `x`, type the following on the command line:

```
rosie hello --prefix=x
```

It should return a greeting, e.g. Hello user.

## 1.5 Deploying Configuration Metadata

You may want to deploy [Configuration Metadata](#) (page 206) for projects using Rose in a globally readable location at your site, so that they can be easily accessed by users when using Rose utilities such as `rose config-edit` (page 250) or `rose macro` (page 254).

If the source tree of a project is version controlled under a trusted Subversion repository, it is possible to automatically deploy their configuration metadata. Assuming that the projects follow our recommendation and store Rose configuration metadata under the `rose-meta/` directory of their source tree, you can:

- Check out a working copy for each sub-directory under the `rose-meta/` directory.
- Set up a crontab job to regularly update the working copies.

For example, suppose you want to deploy Rose [Configuration Metadata](#) (page 206) under `/etc/rose-meta/` at your site. You can do:

```

# Deployment location
DEST='/etc/rose-meta'
cd "${DEST}"

# Assume only Rose metadata configuration directories under "rose-meta/"
URL1='https://somehost/foo/main/trunk/rose-meta'
URL2='https://anotherhost/bar/main/trunk/rose-meta'
# ...

# Checkout a working copy for each metadata configuration directory
for URL in "${URL1}" "${URL2}"; do
    for NAME in $(svn ls "${URL}"); do
        svn checkout -q "${URL}/${NAME}"
    done
done

# Set up a crontab job to update the working copies, e.g. every 10 minutes
crontab -l || true >'crontab.tmp'
{
    echo '# Update Rose configuration metadata every 10 minutes'
    echo "*/10 * * * * svn update -q ${DEST}/*"
} >>'crontab.tmp'
crontab 'crontab.tmp'
rm 'crontab.tmp'

# Finally add the root level "meta-path" setting to site's "rose.conf"
# E.g. if Rose is installed under "/opt/rose/":
{
    echo '[]'
    echo "meta-path=${DEST}"
} >>'/opt/rose/etc/rose.conf'

```

**Tip:** See also [Appendix: Metadata Location](#) (page 216).

## 1.6 Configuring Rose Bush

Rose Bush provides an intranet web service at your site for users to view their suite logs using a web browser. Depending on settings at your site, you may or may not be able to set up this service.

You can start an ad-hoc Rose Bush web server by running:

```
setsid /path/to/rose/bin/rose bush start 0</dev/null 1>/dev/null 2>&1 &
```

You will find the access and error logs under `~/.metomi/rose-bush*`.

Alternatively you can run the Rose Bush web service under Apache mod\_wsgi. To do this you will need to set up an Apache module configuration file (typically in `/etc/httpd/conf.d/rose-wsgi.conf`) containing the following (with the paths set appropriately):

```
WSGIPath /path/to/rose/lib/python
WSGIScriptAlias /rose-bush /path/to/rose/lib/python/rose/bush.py
```

Use the Apache log at e.g. `/var/log/httpd/` to debug problems. See also [Configuring a Rosie Server](#) (page 8).

## 1.7 Configuring a Rosie Server

You should only need to configure and run your own Rosie service if you do not have access to Rosie services on the Internet, or if you need a private Rosie service for your site. Depending on settings at your site, you may or may not be able to set up this service.

You will need to select a machine to host the Subversion repositories. This machine will also host the web server and databases.

Login to your host, create one or more Subversion FSFS ([https://en.wikipedia.org/wiki/Apache\\_Subversion#FSFS](https://en.wikipedia.org/wiki/Apache_Subversion#FSFS)) repositories.

If you want to use FCM for your version control, you should set a special property on the repository to allow branching and merging with FCM in the Rosie convention. For example, if your repository is served from `HOST_AND_PATH` (e.g. `myhost001/svn-repos`) with given repository base name `NAME` (e.g. `roses_foo`), change into a new directory and enter the following commands:

```
svn co -q "svn://${HOST_AND_PATH}/${NAME}/"
svn ps fcm:layout -F - "${NAME}" << '__FCM_LAYOUT__'
depth-project = 5
depth-branch = 1
depth-tag = 1
dir-trunk = trunk
dir-branch =
dir-tag =
level-owner-branch =
level-owner-tag =
template-branch =
template-tag =
__FCM_LAYOUT__
svn ci -m 'fcm:layout: defined.' "${NAME}"
rm -fr "${NAME}"
```

Add the following hook scripts to the repository:

- pre-commit:

```
#!/bin/bash
exec <path-to-rose>/sbin/roса svn-pre-commit "$@"
```

- post-commit:

```
#!/bin/bash
exec <path-to-rose>/sbin/roса svn-post-commit "$@"
```

You should replace `/path/to/rose/` with the location of your Rose installation.

Make sure the hook scripts are executable.

The `roса svn-post-commit` command in the `post-commit` hook is used to populate a database with the suite discovery information as suites are committed to the repository. Edit the `rose.conf[rosie-db]` (page 204) settings to point to your host machine and provide relevant paths such as the location for your repository and database.

Once you have done that, create the Rosie database by running:

```
/path/to/rose/sbin/roса db-create
```

Make sure that the account that runs the repository hooks has read/write access to the database and database directory.

You can test that everything is working using the built-in web server. Edit the `rose.conf[rosie-disco]` (page 204) settings to configure the web server's log directory and port number. Start the web server by running:

```
setsid /path/to/rose/bin/rosie disco start 0</dev/null 1</dev/null 2>&1 &
```

Check that the server is up and running using `curl` or a local web browser. E.g. If you have configured the server's port to be 1234, you can do:

```
curl -I http://localhost:1234/
```

It should return a HTTP code 200.

Alternatively you can run the Rosie web service under Apache `mod_wsgi`. To do this you will need to set up an Apache module configuration file (typically in `/etc/httpd/conf.d/rose-wsgi.conf`) containing the following (with the paths set appropriately):

```
WSGIPath /path/to/rose/lib/python
WSGIScriptAlias /rosie /path/to/rose/lib/python/rozie/ws.py
```

Use the Apache log at e.g. `/var/log/httpd/` to debug problems. See also [Configuring Rose Bush](#) (page 7).

Hopefully, you should now have a working Rosie service server. Configure the client settings by editing the `rose.conf[rosie-id]` (page 204) settings. If you are using the built-in web server, you should ensure that you include the port number in the URL. E.g.:

```
[rosie-id]
prefix-ws.foo=http://127.0.0.1:1234/foo
```

You should now be able to talk to the Rosie web service server via the Rosie web service client. Test by doing:

```
rosie hello
```

To test that everything is connecting together, create your first suite in the repository by doing:

```
rosie create
```

which will create the first suite in your repository, with an ID ending in aa000 - e.g. foo-aa000. Locate it by running:

```
rosie lookup 000
```

### 1.7.1 ROSIE special suite

You can define a special suite in each Rosie repository that provides some additional repository-specific data and metadata. The suite ID will end with ROSIE - e.g. foo-ROSIE.

This can be created by running `rosie create --meta-suite`.

### 1.7.2 Creating a Known Keys File

You can extend the list of search keys used in the Rosie discovery interfaces (such as `rosie go`). Create a text file at the root of a Rosie suite working copy called `rosie-keys`.

Add a space-delimited list of search keys into the file - for example:

```
sub-project experiment model
```

Run `fcm add -c` and `fcm commit`. After the commit, these will be added to the list of Rosie interface search keys.

You can continue to modify the list by changing the file contents and committing.



## GETTING STARTED

If you are working with a new installation of Rose you should look at the [Site And User Configuration](#) (page 198). Rose combines the settings in the site configuration and the user configuration at run time. You can view the resultant configuration by issuing the command:

```
rose config
```

Rose should work out of the box if it is configured correctly at your site.

### 2.1 Text Editor

- The default external text editor used by *GUIs* is gedit.
- The default external text editor used by *CLI* commands is the value of the VISUAL or EDITOR environment variable, or vi if neither environment variable is set.

To change the default editor change the following settings in the user configuration file `~/.metomi/rose.conf`:

**[external]geditor** The external text editor used by GUIs

**[external]editor** The external text editor used by CLI commands

For emacs and most text editors, you can do something like:

```
[external]
editor=emacs
geditor=emacs
```

For any text editor command that normally forks and detaches from the shell it is started in, you should use an option to ensure that the text editor runs in the foreground to allow Rose to wait for the edit session to finish. E.g. for gvim, you will do:

```
[external]
editor=gvim -f
geditor=gvim -f
```

### 2.2 Editor Syntax Highlighting

There are gedit, kate, vim, and emacs plugins for syntax highlighting of Rose configuration files, located within the Rose installation:

- etc/rose-conf.lang
- etc/rose-conf.xml
- etc/rose-conf.vim

- `etc/rose-conf-mode.el`

The plugins contain setup instructions within.

Additionally there is a [Pygments](http://pygments.org) (<http://pygments.org>) lexer located in `sphinx/ext/rose_lang.py`.

---

**Hint:** You can locate your Rose installation using:

```
rose version --long
```

---

## 2.3 Bash Auto-Completion

There is a Rose bash completion script that you can source to enhance the Rose command line interface within an interactive Bash shell.

The script allows you to tab-complete Rose commands, options, and arguments.

You can find the script in the Rose installation `etc/rose-bash-completion`. The file contains the instructions for using it.

## 2.4 Configuring Cylc

See the “Installation” and “User Config File” sections of the Cylc User Guide (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

**Warning:** Do not modify the default values of the following cylc settings:

- `[hosts] [HOST]run directory`
- `[hosts] [HOST]work directory`

Equivalent functionalities are provided by the `rose.conf[rose-suite-run]root-dir` (page 203) settings in the Rose site/user configuration.

## CYLC TUTORIAL

Cylc is a workflow engine for running suites of inter-dependent jobs.



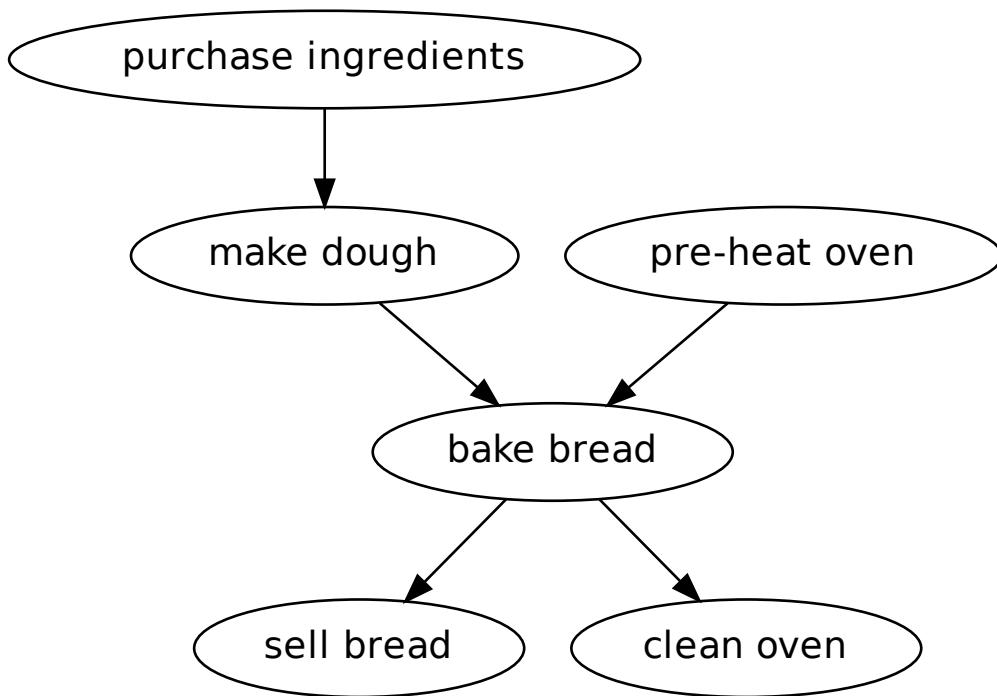
This section will cover the Cylc framework and writing basic Cylc suites.

### 3.1 Introduction

#### 3.1.1 What Is A Workflow?

In research, business and other fields we may have processes that we repeat in the course of our work. At its simplest a workflow is a set of steps that must be followed in a particular order to achieve some end goal.

We can represent each “step” in a workflow as a oval and the order with arrows.



### 3.1.2 What Is Cyc?

Cyclc (pronounced silk) is a workflow engine, a system that automatically executes tasks according to their schedules and dependencies. In a Cyclc workflow each step is a computational task, a script to execute. Cyclc runs each task as soon as it is appropriate to do so. Cyclc was originally developed at NIWA (The National Institute of Water and Atmospheric Research - New Zealand) for running their weather forecasting workflows. Cyclc is now developed by an international partnership including members from NIWA and the Met Office (UK). Though initially developed for meteorological purposes Cyclc is a general purpose tool as applicable in business as in scientific research.

## 3.2 Scheduling

This section looks at how to write workflows in cyclc.

### 3.2.1 Graphing

In this section we will cover writing basic workflows in cyclc.

#### The `suite.rc` File Format

We refer to a Cyclc workflow as a *Cyclc suite*. A Cyclc suite is a directory containing a `suite.rc` file. This configuration file is where we define our workflow. The `suite.rc` file uses a nested [INI](#) ([https://en.wikipedia.org/wiki/INI\\_file](https://en.wikipedia.org/wiki/INI_file))-based format:

- Comments start with a # character.

- Settings are written as `key = value` pairs.
- Settings can be contained within sections.
- Sections are written inside square brackets i.e. `[section-name]`.
- Sections can be nested, by adding an extra square bracket with each level, so a sub-section would be written `[ [ sub-section ] ]`, a sub-sub-section `[ [ [ sub-sub-section ] ] ]`, and so on.

```
# Comment
[section]
    key = value
    [[sub-section]]
        another-key = another-value # Inline comment
        yet-another-key = """
A
Multi-line
String
"""
```

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

- `[section]` - refers to the entire section.
- `[section]key` - refers to a setting within the section.
- `[section]key=value` - expresses the value of the setting.
- `[section] [sub-section]another-key`. Note we only use one set of square brackets with nested sections.

---

**Tip:** It is advisable to indent `suite.rc` files. This indentation, however, is ignored when the file is parsed so settings must appear before sub-sections.

```
[section]
    key = value # This setting belongs to the section.
    [[sub-section]]
        key = value # This setting belongs to the sub-section.

# This setting belongs to the sub-section as indentation is ignored.
# Always write settings before defining any sub-sections!
    key = value
```

---

## Note

In the `suite.rc` file format duplicate sections are additive, that is to say the following two examples are equivalent:

```
[a]
    c = C
[b]
    d = D
[a]
    e = E
```

```
[a]
    c = C
    e = E
[b]
    d = D
```

Settings, however, are not additive meaning that a duplicate setting will override an earlier value. The following two examples are also equivalent:

```
a = foo  
a = bar
```

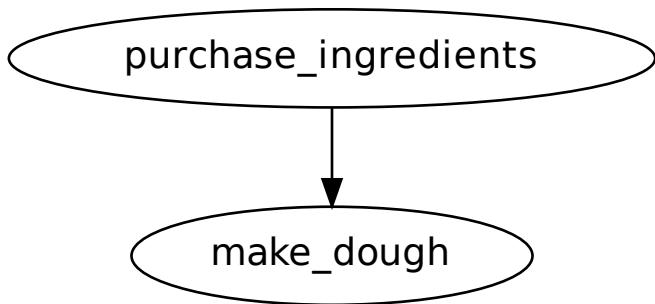
```
a = bar
```

## Graph Strings

In Cyc we consider workflows in terms of *tasks* and *dependencies*.

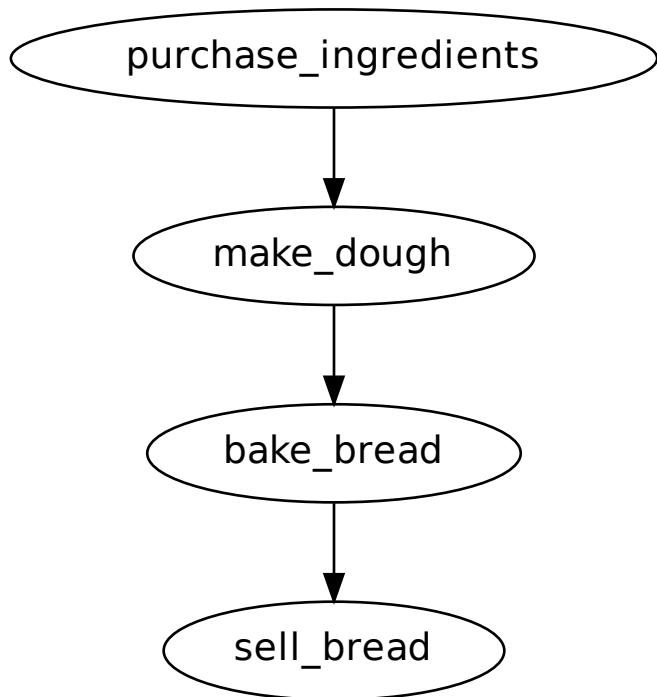
Tasks are represented as words and dependencies as arrows ( $=>$ ), so the following text defines two tasks where `make_dough` is dependent on `purchase_ingredients`:

```
purchase_ingredients => make_dough
```



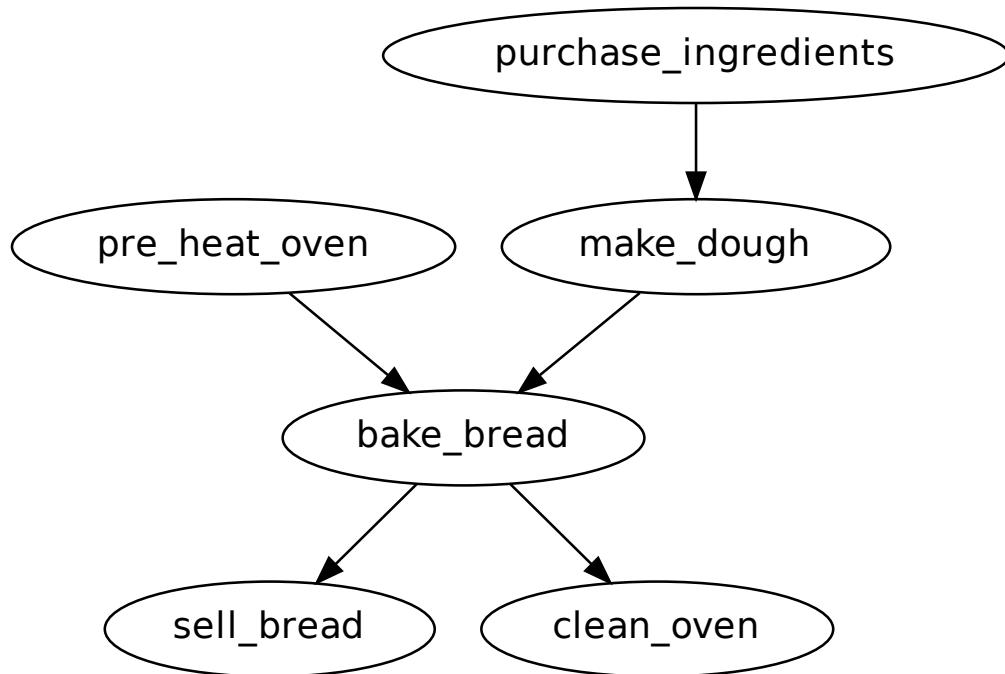
In a Cyc workflow this would mean that `make_dough` would only run when `purchase_ingredients` has succeeded. These *dependencies* can be chained together:

```
purchase_ingredients => make_dough => bake_bread => sell_bread
```



This line of text is referred to as a *graph string*. These graph strings can be combined to form more complex workflows:

```
purchase_ingredients => make_dough => bake_bread => sell_bread  
pre_heat_oven => bake_bread  
bake_bread => clean_oven
```



Graph strings can also contain “and” (`&`) and “or” (`|`) operators, for instance the following lines are equivalent to the ones just above:

```

purchase_ingredients => make_dough
pre_heat_oven & make_dough => bake_bread => sell_bread & clean_oven
  
```

Collectively these *graph strings* are referred to as a *graph*.

### Note

The order in which lines appear in the graph section doesn't matter, for instance the following examples are the same as each other:

```

foo => bar
bar => baz
  
```

```

bar => baz
foo => bar
  
```

### Cyclc Graphs

In a *Cyclc suite* the *graph* is stored under the `[scheduling][dependencies]` graph setting, i.e:

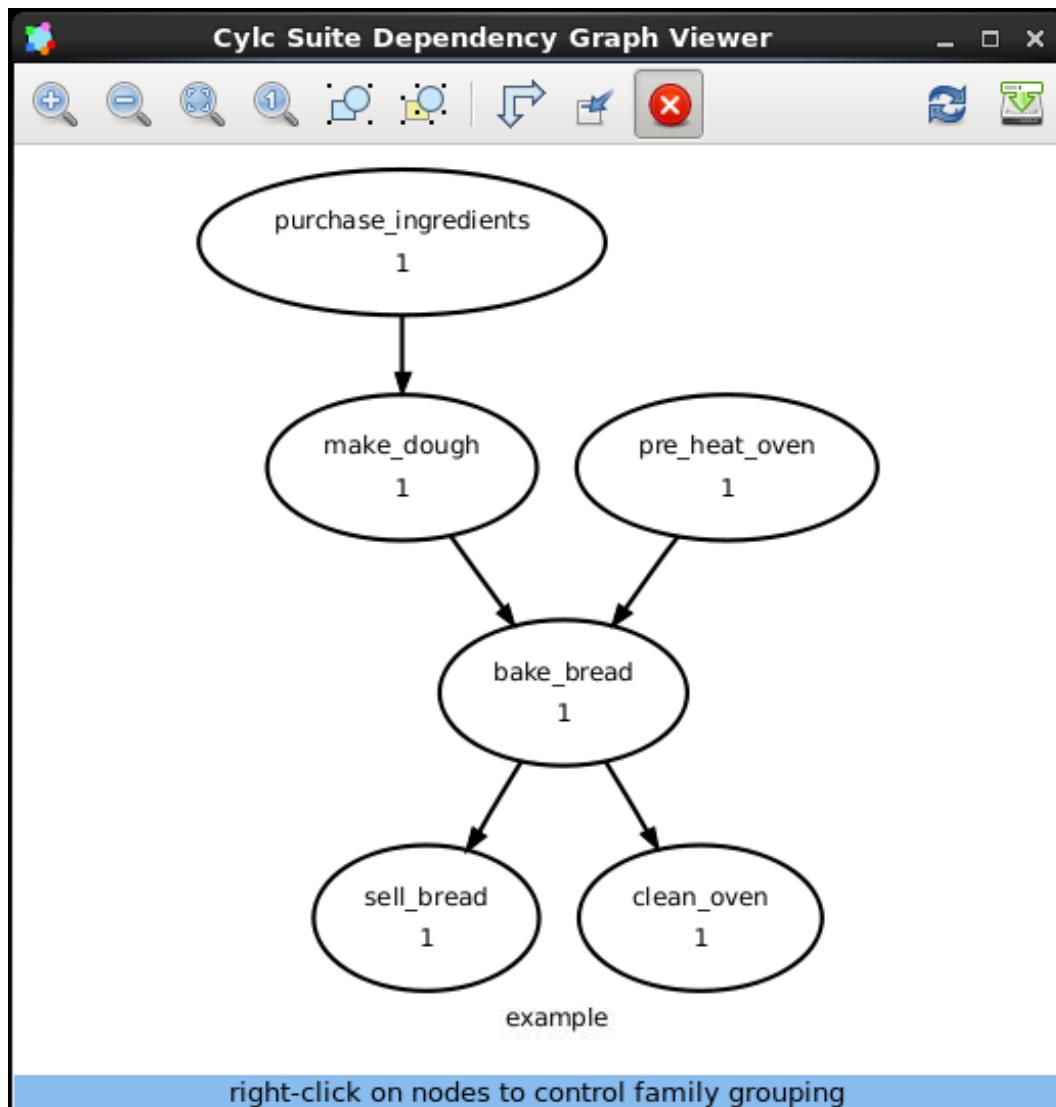
```

[scheduling]
  [[dependencies]]
  graph =
    purchase_ingredients => make_dough
    pre_heat_oven & make_dough => bake_bread => sell_bread & clean_oven
    """
  
```

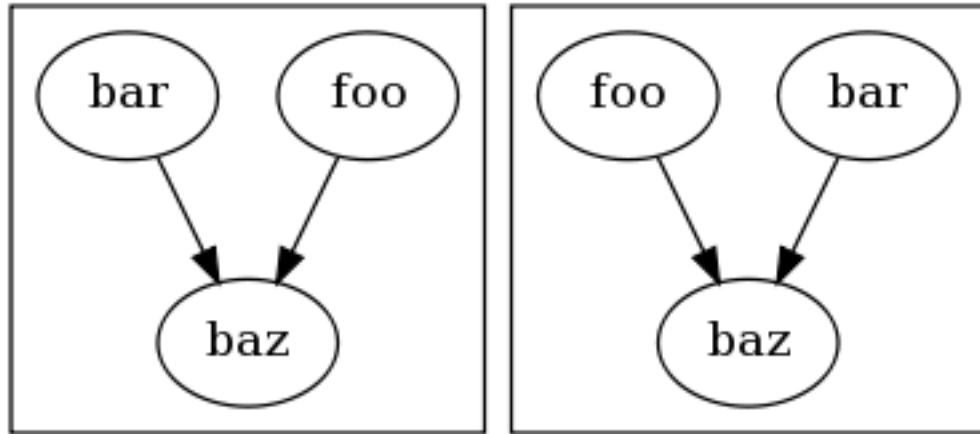
This is a minimal *Cylc suite*, in which we have defined a *graph* representing a workflow for Cylc to run. We have not yet provided Cylc with the scripts or binaries to run for each task. This will be covered later in the *runtime tutorial* (page 39).

Cylc provides a GUI for visualising *graphs*. It is run on the command line using the `cylc graph <path>` command where the path `path` is to the `suite.rc` file you wish to visualise.

When run, `cylc graph` will display a diagram similar to the ones you have seen so far. The number 1 which appears below each task is the *cycle point*. We will explain what this means in the next section.



**Hint:** A graph can be drawn in multiple ways, for instance the following two examples are equivalent:



The graph drawn by `cylc graph` may vary slightly from one run to another but the tasks and dependencies will always be the same.

---

## Practical

In this practical we will create a new Cylc suite and write a graph for it to use.

### 1. Create a Cylc suite.

A Cylc suite is just a directory containing a `suite.rc` file.

If you don't have one already, create a `cylc-run` directory in your user space i.e:

```
~/cylc-run
```

Within this directory create a new folder called `graph-introduction`, which is to be our *suite directory*. Move into it:

```
mkdir ~/cylc-run/graph-introduction  
cd ~/cylc-run/graph-introduction
```

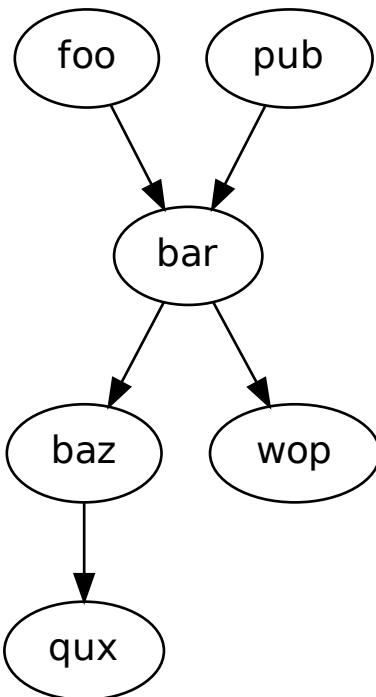
Inside this directory create a `suite.rc` file and paste in the following text:

```
[scheduling]  
  [[dependencies]]  
    graph = """  
      # Write graph strings here!  
    """
```

### 2. Write a graph.

We now have a blank Cylc suite, next we need to define a workflow.

Edit your `suite.rc` file to add graph strings representing the following graph:



### 3. Use cylc graph to visualise the workflow.

Once you have written some graph strings try using `cylc graph` to display the workflow. Run the following command:

```
cylc graph .
```

---

#### Note

`cylc graph` takes the path to the suite as an argument. As we are inside the *suite directory* we can run `cylc graph ..`

---

If the results don't match the diagram above try going back to the `suite.rc` file and making changes.

---

**Tip:** In the top right-hand corner of the `cylc graph` window there is a refresh button which will reload the GUI with any changes you have made.




---

#### Solution

There are multiple correct ways to write this graph. So long as what you see in `cylc graph` matches the above diagram then you have a correct solution.

Two valid examples:

```
foo & pub => bar => baz & wop
baz => qux
```

```
foo => bar => baz => qux
pub => bar => wop
```

The whole suite should look something like this:

```
[scheduling]
[[dependencies]]
graph =
    foo & pub => bar => baz & wop
    baz => qux
    ...
```

### 3.2.2 Basic Cycling

In this section we will look at how to write *cycling* (repeating) workflows.

#### Repeating Workflows

Often, we will want to repeat the same workflow multiple times. In Cylc this “repetition” is called *cycling* and each repetition of the workflow is referred to as a *cycle*.

Each *cycle* is given a unique label. This is called a *cycle point*. For now these *cycle points* will be integers (*they can also be dates as we will see in the next section*).

To make a workflow repeat we must tell Cylc three things:

**The recurrence** How often we want the workflow to repeat.

**The initial cycle point** At what cycle point we want to start the workflow.

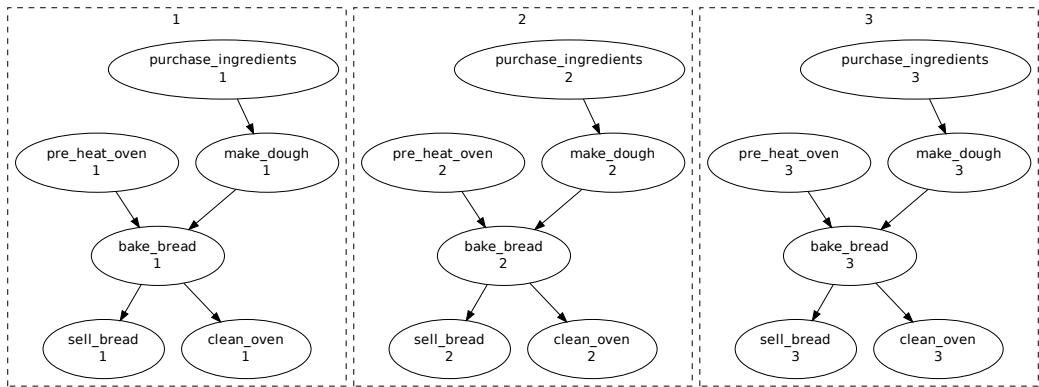
**The final cycle point** Optionally we can also tell Cylc what cycle point we want to stop the workflow.

Let’s take the bakery example from the previous section. Bread is produced in batches so the bakery will repeat this workflow for each batch of bread they bake. We can make this workflow repeat with the addition of three lines:

```
[scheduling]
+   cycling mode = integer
+   initial cycle point = 1
[[dependencies]]
+   [[[P1]]]
graph =
    purchase_ingredients => make_dough
    pre_heat_oven & make_dough => bake_bread => sell_bread & clean_
    ↵oven
    ...
```

- The `cycling mode = integer` setting tells Cylc that we want our *cycle points* to be numbered.
- The `initial cycle point = 1` setting tells Cylc to start counting from 1.
- `P1` is the *recurrence*. The `graph` within the `[[ [ P1 ] ]]` section will be repeated at each *cycle point*.

The first three *cycles* would look like this, with the entire workflow repeated at each cycle point:



Note the numbers under each task which represent the *cycle point* each task is in.

## Inter-Cycle Dependencies

We've just seen how to write a workflow that repeats every *cycle*.

Cyc runs tasks as soon as their dependencies are met so cycles are not necessarily run in order. This could cause problems, for instance we could find ourselves pre-heating the oven in one cycle whilst we are still cleaning it in another.

To resolve this we must add *dependencies between the cycles*. We do this by adding lines to the *graph*. Tasks in the previous cycle can be referred to by suffixing their name with `[-P1]`, for example. So to ensure the `clean_oven` task has been completed before the start of the `pre_heat_oven` task in the next cycle, we would write the following dependency:

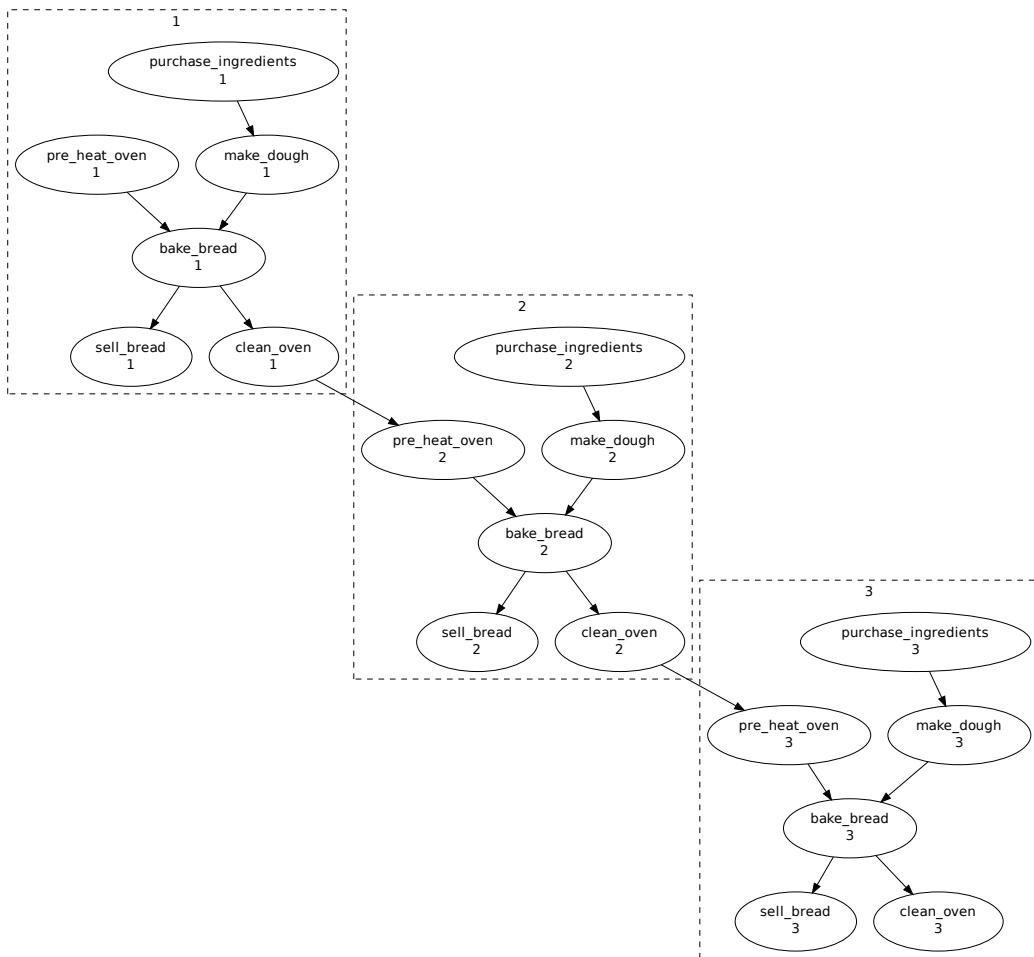
```
clean_oven[-P1] => pre_heat_oven
```

This dependency can be added to the suite by adding it to the other graph lines:

```
[scheduling]
cycling mode = integer
initial cycle point = 1
[[dependencies]]
[[[P1]]]
graph = """
purchase_ingredients => make_dough
pre_heat_oven & make_dough => bake_bread => sell_bread & clean_
oven
+     clean_oven[-P1] => pre_heat_oven
"""

```

The resulting suite would look like this:



Adding this dependency “strings together” the cycles, forcing them to run in order. We refer to dependencies between cycles as *inter-cycle dependencies*.

In the dependency the `[-P1]` suffix tells CycL that we are referring to a task in the previous cycle. Equally `[-P2]` would refer to a task two cycles ago.

Note that the `purchase_ingredients` task has no arrows pointing at it meaning that it has no dependencies. Consequently the `purchase_ingredients` tasks will all run straight away. This could cause our bakery to run into cash-flow problems as they would be purchasing ingredients well in advance of using them.

To solve this, but still make sure that they never run out of ingredients, the bakery wants to purchase ingredients two batches ahead. This can be achieved by adding the following dependency:

```

[scheduling]
cycling mode = integer
initial cycle point = 1
[[dependencies]]
[[[P1]]]
graph =
    purchase_ingredients => make_dough
    pre_heat_oven & make_dough => bake_bread => sell_bread & clean_
    ↵oven
+
    clean_oven[-P1] => pre_heat_oven
    sell_bread[-P2] => purchase_ingredients

```

(continues on next page)

(continued from previous page)

"""

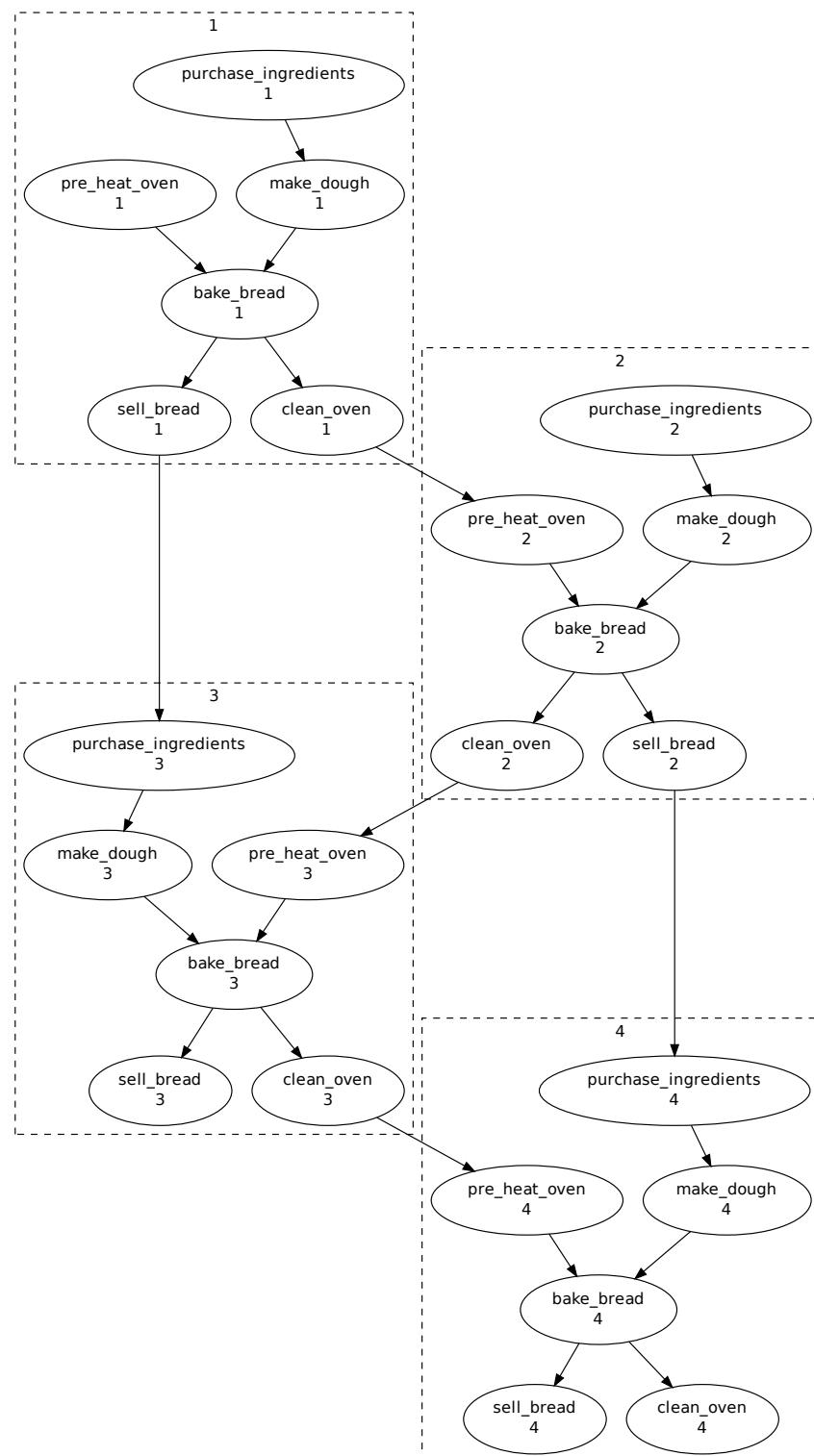
This dependency means that the `purchase_ingredients` task will run after the `sell_bread` task two cycles before.

---

**Note:** The `[-P2]` suffix is used to reference a task two cycles before. For the first two cycles this doesn't make sense as there was no cycle two cycles before, so this dependency will be ignored.

Any inter-cycle dependencies stretching back to before the *initial cycle point* will be ignored.

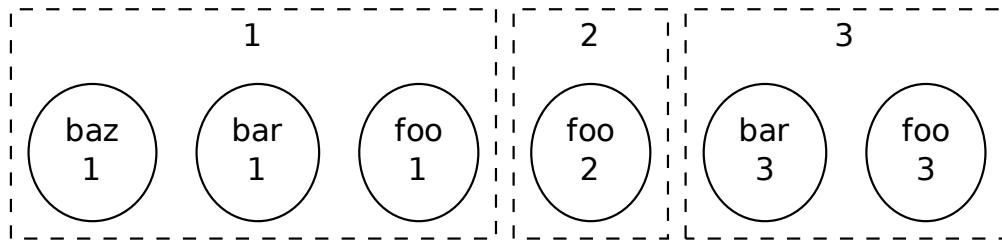
---



## Recurrence Sections

In the previous examples we made the workflow repeat by placing the graph within the `[[ [P1] ]]` section. Here P1 is a *recurrence* meaning repeat every cycle, where P1 means every cycle, P2 means every *other* cycle, and so on. To build more complex workflows we can use multiple recurrences:

```
[scheduling]
cycling mode = integer
initial cycle point = 1
[[dependencies]]
  [[P1]] # Repeat every cycle.
    graph = foo
  [[P2]] # Repeat every second cycle.
    graph = bar
  [[P3]] # Repeat every third cycle.
    graph = baz
```



By default recurrences start at the *initial cycle point*, however it is possible to make them start at an arbitrary cycle point. This is done by writing the cycle point and the recurrence separated by a forward slash (/), e.g. 5/P3 means repeat every third cycle starting *from* cycle number 5.

The start point of a recurrence can also be defined as an offset from the *initial cycle point*, e.g. +P5/P3 means repeat every third cycle starting 5 cycles *after* the initial cycle point.

---

## Practical

**In this practical we will take the suite we wrote in the previous section and turn it into a cycling suite.**

If you have not completed the previous practical use the following code for your `suite.rc` file.

```
[scheduling]
[[dependencies]]
graph = """
  foo & pub => bar => baz & wop
  baz => qux
"""
"""
```

1. **Create a new suite.**

Within your `~/cylc-run/` directory create a new (sub-)directory called `integer-cycling` and move into it:

```
mkdir ~/cylc-run/integer-cycling
cd ~/cylc-run/integer-cycling
```

Copy the above code into a `suite.rc` file in that directory.

## 2. Make the suite cycle.

Add in the following lines.

```
[scheduling]
+   cycling mode = integer
+   initial cycle point = 1
  [[dependencies]]
+   [[[P1]]]
     graph = """
       foo & pub => bar => baz & wop
       baz => qux
     """

```

## 3. Visualise the suite.

Try visualising the suite using `cylc graph`.

```
cylc graph .
```

**Tip:** You can get Cycl graph to draw dotted boxes around the cycles by clicking the “Organise by cycle point” button on the toolbar:



**Tip:** By default `cylc graph` displays the first three cycles of a suite. You can tell `cylc graph` to visualise the cycles between two points by providing them as arguments, for instance the following example would show all cycles between 1 and 5 (inclusive):

```
cylc graph . 1 5 &
```

## 4. Add another recurrence.

Suppose we wanted the `qux` task to run every *other* cycle as opposed to every cycle. We can do this by adding another recurrence.

Make the following changes to your `suite.rc` file.

```
[scheduling]
  cycling mode = integer
  initial cycle point = 1
  [[dependencies]]
    [[[P1]]]
      graph = """
        foo & pub => bar => baz & wop
      -      baz => qux
      """
+
    [[[P2]]]
+
      graph = """
+
      baz => qux
+
      """

```

Use `cylc graph` to see the effect this has on the workflow.

## 5. Inter-cycle dependencies.

Next we need to add some inter-cycle dependencies. We are going to add three inter-cycle dependencies:

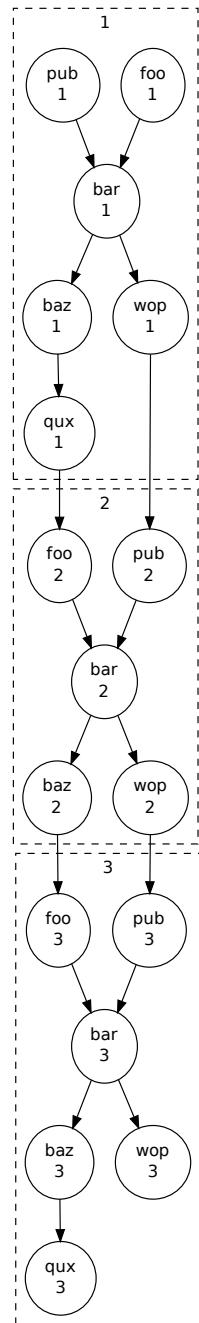
- (a) Between `wop` from the previous cycle and `pub`.
- (b) Between `baz` from the previous cycle and `foo` *every odd cycle*.
- (c) Between `qux` from the previous cycle and `foo` *every even cycle*.

Have a go at adding inter-cycle dependencies to your `suite.rc` file to make your workflow match the diagram below.

---

**Hint:**

- `P2` means every odd cycle.
  - `2/P2` means every even cycle.
-



---

### Solution

```

[scheduling]
cycling mode = integer
initial cycle point = 1
[[[dependencies]]
[[[P1]]]
graph = """

```

(continues on next page)

(continued from previous page)

```

foo & pub => bar => baz & wop
wop[-P1] => pub # (1)
"""

[[[P2]]]
graph =
    baz => qux
    baz[-P1] => foo # (2)
"""

[[[+P2/P2]]]
graph =
    qux[-P1] => foo # (3)
"""

```

### 3.2.3 Date-Time Cycling

In the last section we looked at writing an *integer cycling* workflow, one where the *cycle points* are numbered.

Typically workflows are repeated at a regular time interval, say every day or every few hours. To make this easier Cyc has a date-time cycling mode where the *cycle points* use date and time specifications rather than numbers.

#### Reminder

*Cycle points* are labels. Cyc runs tasks as soon as their dependencies are met so cycles do not necessarily run in order.

## ISO8601

In Cyc, dates, times and durations are written using the *ISO8601* format - an international standard for representing dates and times.

### Date-Times

In ISO8601, datetimes are written from the largest unit to the smallest (i.e: year, month, day, hour, minute, second in succession) with the T character separating the date and time components. For example, midnight on the 1st of January 2000 is written 20000101T000000.

For brevity we may omit seconds (and minutes) from the time i.e: 20000101T0000 (20000101T00).

For readability we may add hyphen (-) characters between the date components and colon (:) characters between the time components, i.e: 2000-01-01T00:00. This is the “extended” format.

Time-zone information can be added onto the end. UTC is written Z, UTC+1 is written +01, etc. E.G: 2000-01-01T00:00Z.



**Warning:** The “basic” (purely numeric except for T) and “extended” (written with hyphens and colons) formats cannot be mixed. For example the following date-times are invalid:

```
2000-01-01T0000
20000101T00:00
```

## Durations

In ISO8601, durations are prefixed with a P and are written with a character following each unit:

- Y for year.
- M for month.
- D for day.
- W for week.
- H for hour.
- M for minute.
- S for second.

As with datetimes the components are written in order from largest to smallest and the date and time components are separated by the T character. E.G:

- P1D: one day.
- PT1H: one hour.
- P1DT1H: one day and one hour.
- PT1H30M: one and a half hours.
- P1Y1M1DT1H1M1S: a year and a month and a day and an hour and a minute and a second.

## Date-Time Recurrences

In *integer cycling*, suites' recurrences are written P1, P2, etc.

In *date-time cycling* there are two ways to write recurrences:

1. Using ISO8601 durations (e.g. P1D, PT1H).
2. Using ISO8601 date-times with inferred recurrence.

## Inferred Recurrence

A recurrence can be inferred from a date-time by omitting digits from the front. For example, if the year is omitted then the recurrence can be inferred to be annual. E.G:

```
2000-01-01T00    # Datetime - midnight on the 1st of January 2000.  
  
01-01T00        # Every year on the 1st of January.  
    01T00        # Every month on the first of the month.  
        T00        # Every day at midnight.  
            T-00      # Every hour at zero minutes past (every hour on the hour).
```

---

**Note:** To omit hours from a date time we must place a – after the T character.

---

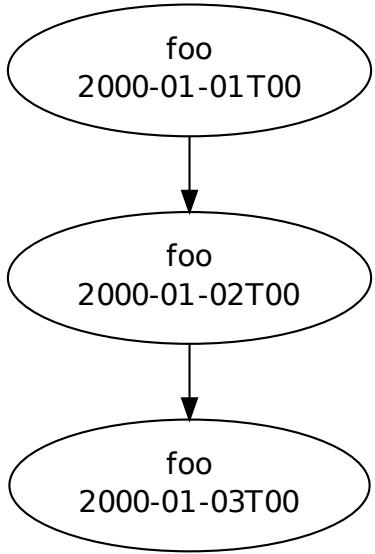
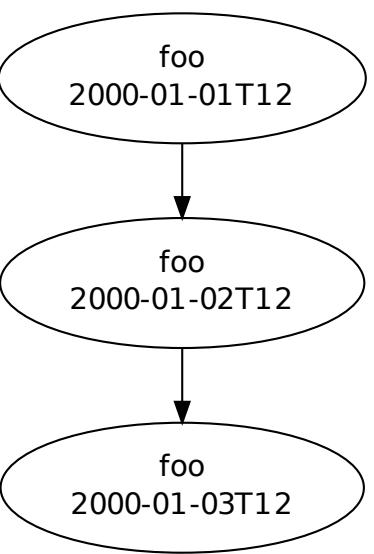
## Recurrence Formats

As with integer cycling, recurrences start, by default, at the *initial cycle point*. We can override this in one of two ways:

1. By defining an arbitrary cycle point (datetime/recurrence):
  - 2000/P1Y: every year starting with the year 2000.
  - 2000-01-01T00/T00: every day at midnight starting on the 1st of January 2000
  - 2000-01-01T12/T00: every day at midnight starting on the first midnight after the 1st of January at 12:00 (i.e. 2000-01-02T00).
2. By defining an offset from the initial cycle point (offset/recurrence). This offset is an ISO8601 duration preceded by a plus character:
  - +P1Y/P1Y: every year starting one year after the initial cycle point.
  - +PT1H/T00: every day starting on the first midnight after the point one hour after the initial cycle point.

## Durations And The Initial Cycle Point

When using durations, beware that a change in the initial cycle point might produce different results for the recurrences.

<pre>[scheduling] initial cycle point = 2000-01-01T00 [[dependencies]] [[[P1D]]] graph = foo[-P1D] =&gt; foo</pre>	<pre>[scheduling] initial cycle point = 2000-01-01T12 [[dependencies]] [[[P1D]]] graph = foo[-P1D] =&gt; foo</pre>
 <pre> graph TD     A((foo 2000-01-01T00)) --&gt; B((foo 2000-01-02T00))     B --&gt; C((foo 2000-01-03T00))   </pre>	 <pre> graph TD     A((foo 2000-01-01T12)) --&gt; B((foo 2000-01-02T12))     B --&gt; C((foo 2000-01-03T12))   </pre>

We could write the recurrence “every midnight” independent from the initial cycle point by:

- Use an *inferred recurrence* (page 32) instead (i.e. T00).
- Overriding the recurrence start point (i.e. T00/P1D)
- Using the [scheduling]initial cycle point constraints setting to constrain the initial cycle point (e.g. to a particular time of day). See the Cyc User Guide (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>) for details.

## The Initial & Final Cycle Points

There are two special recurrences for the initial and final cycle points:

- R1: repeat once at the initial cycle point.
- R1/P0Y: repeat once at the final cycle point.

## Inter-Cycle Dependencies

Inter-cycle dependencies are written as ISO8601 durations, e.g:

- foo [-P1D]: the task foo from the cycle one day before.
- bar [-PT1H30M]: the task bar from the cycle 1 hour 30 minutes before.

The initial cycle point can be referenced using a caret character ^, e.g:

- baz [^]: the task baz from the initial cycle point.

## UTC Mode

Due to all of the difficulties caused by time zones, particularly with respect to daylight savings, we typically use UTC (that's the +00 time zone) in Cyc suites.

When a suite uses UTC all of the cycle points will be written in the +00 time zone.

To make your suite use UTC set the [cylc]UTC mode setting to True, i.e:

```
[cylc]
  UTC mode = True
```

## Putting It All Together

Cyc was originally developed for running operational weather forecasting. In this section we will outline a basic (dummy) weather-forecasting suite and explain how to implement it in cylc.

---

**Note:** Technically the suite outlined in this section is a nowcasting (<https://www.metoffice.gov.uk/learning/science/hours-ahead/nowcasting>) suite. We will refer to it as forecasting for simplicity.

---

A basic weather-forecasting workflow consists of three main steps:

### 1. Gathering Observations

We gather observations from different weather stations and use them to build a picture of the current weather. Our dummy weather forecast will get wind observations from four weather stations:

- Belmullet
- Camborne

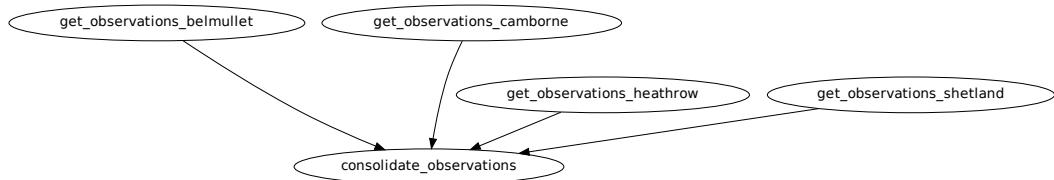
- Heathrow
- Shetland

The tasks which retrieve observation data will be called `get_observations_<site>` where `site` is the name of the weather station in question.

Next we need to consolidate these observations so that our forecasting system can work with them. To do this we have a `consolidate_observations` task.

We will fetch wind observations **every three hours starting from the initial cycle point**.

The `consolidate_observations` task must run after the `get_observations<site>` tasks.



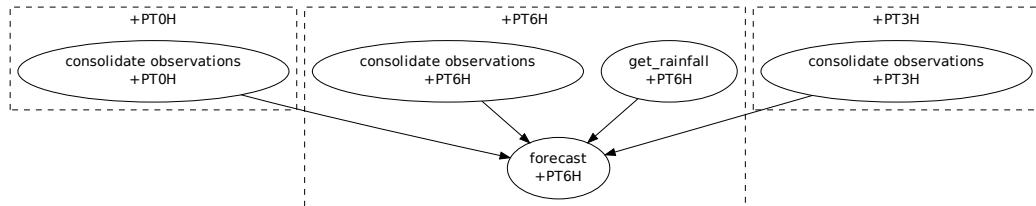
We will also use the UK radar network to get rainfall data with a task called `get_rainfall`.

We will fetch rainfall data **every six hours starting six hours after the initial cycle point**.

## 2. Running computer models to generate forecast data

We will do this with a task called `forecast` which will run **every six hours starting six hours after the initial cycle point**. The `forecast` task will be dependent on:

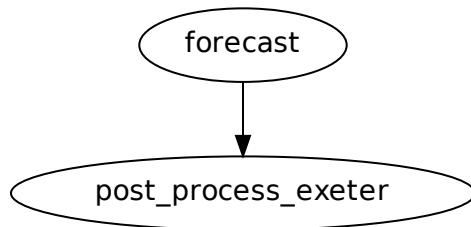
- The `consolidate_observations` task from the previous two cycles as well as from the present cycle.
- The `get_rainfall` task from the present cycle.



## 3. Processing the data output to produce user-friendly forecasts

This will be done with a task called `post_process_<location>` where `location` is the place we want to generate the forecast for. For the moment we will use Exeter.

The `post_process_exeter` task will run **every six hours starting six hours after the initial cycle point** and will be dependent on the `forecast` task.



---

## Practical

In this practical we will create a dummy forecasting suite using date-time cycling.

### 1. Create A New Suite.

Within your `~/cylc-run` directory create a new directory called `datetime-cycling` and move into it:

```
mkdir ~/cylc-run/datetime-cycling  
cd ~/cylc-run/datetime-cycling
```

Create a `suite.rc` file and paste the following code into it:

```
[cylc]  
  UTC mode = True  
[scheduling]  
  initial cycle point = 20000101T00Z  
  [[dependencies]]
```

### 2. Add The Recurrences.

The weather-forecasting suite will require two recurrences. Add sections under the dependencies section for these, based on the information given above.

---

**Hint:** See *Date-Time Recurrences* (page 33).

---

---

## Solution

The two recurrences you need are

- PT3H: repeat every three hours starting from the initial cycle point.
- +PT6H/PT6H: repeat every six hours starting six hours after the initial cycle point.

```
[cylc]  
  UTC mode = True  
[scheduling]  
  initial cycle point = 20000101T00Z  
  [[dependencies]]  
+   [[[PT3H]]]  
+   [[[+PT6H/PT6H]]]
```

### 3. Write The Graphing.

With the help of the graphs and the information above add dependencies to your suite to implement the weather-forecasting workflow.

You will need to consider the inter-cycle dependencies between tasks.

Use cylc graph to inspect your work.

#### Hint

The dependencies you will need to formulate are as follows:

- The consolidate\_observations task is dependent on the get\_observations\_<site> tasks.
- The forecast task is dependent on:
  - the get\_rainfall task;
  - the consolidate\_observations tasks from:
    - \* the same cycle;
    - \* the cycle 3 hours before (-PT3H);
    - \* the cycle 6 hours before (-PT6H).
- The post\_process\_exeter task is dependent on the forecast task.

To launch cylc graph run the command:

```
cylc graph <path/to/suite.rc>
```

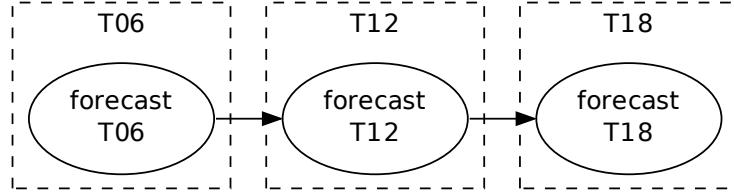
#### Solution

```
[cylc]
    UTC mode = True
[scheduling]
    initial cycle point = 20000101T00Z
    [[dependencies]]
        [[[PT3H]]]
            graph = """
                get_observations_belmullet => consolidate_observations
                get_observations_camborne => consolidate_observations
                get_observations_heathrow => consolidate_observations
                get_observations_shetland => consolidate_observations
            """
        [[[+PT6H/PT6H]]]
            graph = """
                consolidate_observations => forecast
                consolidate_observations[-PT3H] => forecast
                consolidate_observations[-PT6H] => forecast
                get_rainfall => forecast => post_process_exeter
            """
        """
    """

[[[+1]]]
```

### 4. Inter-Cycle Offsets.

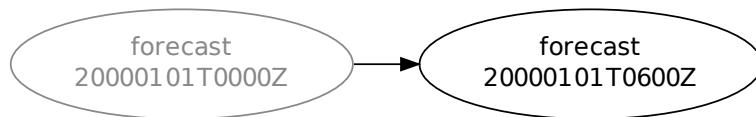
To ensure the forecast tasks for different cycles run in order the forecast task will also need to be dependent on the previous run of forecast.



We can express this dependency as `forecast [-PT6H] => forecast`.

Try adding this line to your suite then visualising it with `cyclc graph`.

You will notice that there is a dependency which looks like this:



Note in particular that the `forecast` task in the 00:00 cycle is grey. The reason for this is that this task does not exist. Remember the `forecast` task runs every six hours **starting 6 hours after the initial cycle point**, so the dependency is only valid from 12:00 onwards. To fix the problem we must add a new dependency section which repeats every six hours **starting 12 hours after the initial cycle point**.

Make the following changes to your suite and the grey task should disappear:

```

[ [[+PT6H/PT6H]] ]
graph =
"""
...
-   forecast [-PT6H] => forecast
"""
+
[ [[+PT12H/PT6H]] ]
+
graph =
"""
+   forecast [-PT6H] => forecast
"""
+

```

### 3.2.4 Further Scheduling

In this section we will quickly run through some of the more advanced features of Cyc's scheduling logic.

#### Qualifiers

So far we have written dependencies like `foo => bar`. This is, in fact, shorthand for `foo:succeed => bar`. It means that the task `bar` will run once `foo` has finished successfully. If `foo` were to fail then `bar` would not run. We will talk more about these *task states* in the [Runtime Section](#).

We refer to the `:succeed` descriptor as a *qualifier*. There are qualifiers for different *task states* e.g:

- :start** When the task has started running.
- :fail** When the task finishes if it fails (produces non-zero return code).
- :finish** When the task has completed (either succeeded or failed).

It is also possible to create your own custom *qualifiers* to handle events within your code (custom outputs).

For more information see the Cycl User Guide (<http://cylc.github.io/cyclc/documentation.html#the-cyclc-user-guide>).

## Clock Triggers

In Cycl, *cycle points* are just labels. Tasks are triggered when their dependencies are met irrespective of the cycle they are in, but we can force cycles to wait for a particular time before running using clock triggers. This is necessary for certain operational and monitoring systems.

For example in the following suite the cycle 2000-01-01T12Z will wait until 11:00 on the 1st of January 2000 before running:

```
[scheduling]
initial cycle point = 2000-01-01T00Z
[[special tasks]]
    clock-trigger = daily(-PT1H)
[[dependencies]]
    [[[T12]]]
        graph = daily    # "daily" will run, at the earliest, one hour
                           # before midday.
```

---

**Tip:** See the *Clock Triggered Tasks* (page 65) tutorial for more information.

---

## Alternative Calendars

By default Cycl uses the Gregorian calendar for *datetime cycling*, but Cycl also supports the 360-day calendar (12 months of 30 days each in a year).

```
[scheduling]
cycling mode = 360day
```

For more information see the Cycl User Guide (<http://cylc.github.io/cyclc/documentation.html#the-cyclc-user-guide>).

## 3.3 Runtime

This section covers:

- Associating tasks with scripts and executables.
- Providing executables with runtime configurations.
- Running Cycl suites.

### 3.3.1 Introduction

So far we have been working with the [scheduling] section. This is where the workflow is defined in terms of *tasks* and *dependencies*.

In order to make the workflow runnable we must associate tasks with scripts or binaries to be executed when the task runs. This means working with the `[runtime]` section which determines what runs, as well as where and how it runs.

## The Task Section

The runtime settings for each task are stored in a sub-section of the `[runtime]` section. E.g. for a task called `hello_world` we would write settings inside the following section:

```
[runtime]
  [[hello_world]]
```

## The script Setting

We tell Cylc *what* to execute when a task is run using the `script` setting.

This setting is interpreted as a bash script. The following example defines a task called `hello_world` which writes `Hello World!` to `stdout` upon execution.

```
[runtime]
  [[hello_world]]
    script = echo 'Hello World!'
```

---

**Note:** If you do not set the `script` for a task then nothing will be run.

---

We can also call other scripts or executables in this way, e.g:

```
[runtime]
  [[hello_world]]
    script = ~/foo/bar/baz/hello_world
```

It is often a good idea to keep our scripts within the Cylc suite directory tree rather than leaving them somewhere else on the system. If you create a `bin` sub-directory within the [suite directory](#) this directory will be added to the path when tasks run, e.g:

Listing 1: bin/hello\_world

```
#!/usr/bin/bash
echo 'Hello World!'
```

Listing 2: suite.rc

```
[runtime]
  [[hello_world]]
    script = hello_world
```

## Tasks And Jobs

When a `task` is “Run” it creates a `job`. The job is a bash file containing the script you have told the task to run along with configuration specifications and a system for trapping errors. It is the `job` which actually gets executed and not the task itself. This “job file” is called the `job script`.

During its life a typical `task` goes through the following states:

**Waiting** `Tasks` wait for their dependencies to be satisfied before running. In the meantime they are in the “Waiting” state.

**Submitted** When a *task's* dependencies have been met it is ready for submission. During this phase the *job script* is created. The *job* is then submitted to the specified batch system. There is more about this in the *next section* (page 47).

**Running** A *task* is in the “Running” state as soon as the *job* is executed.

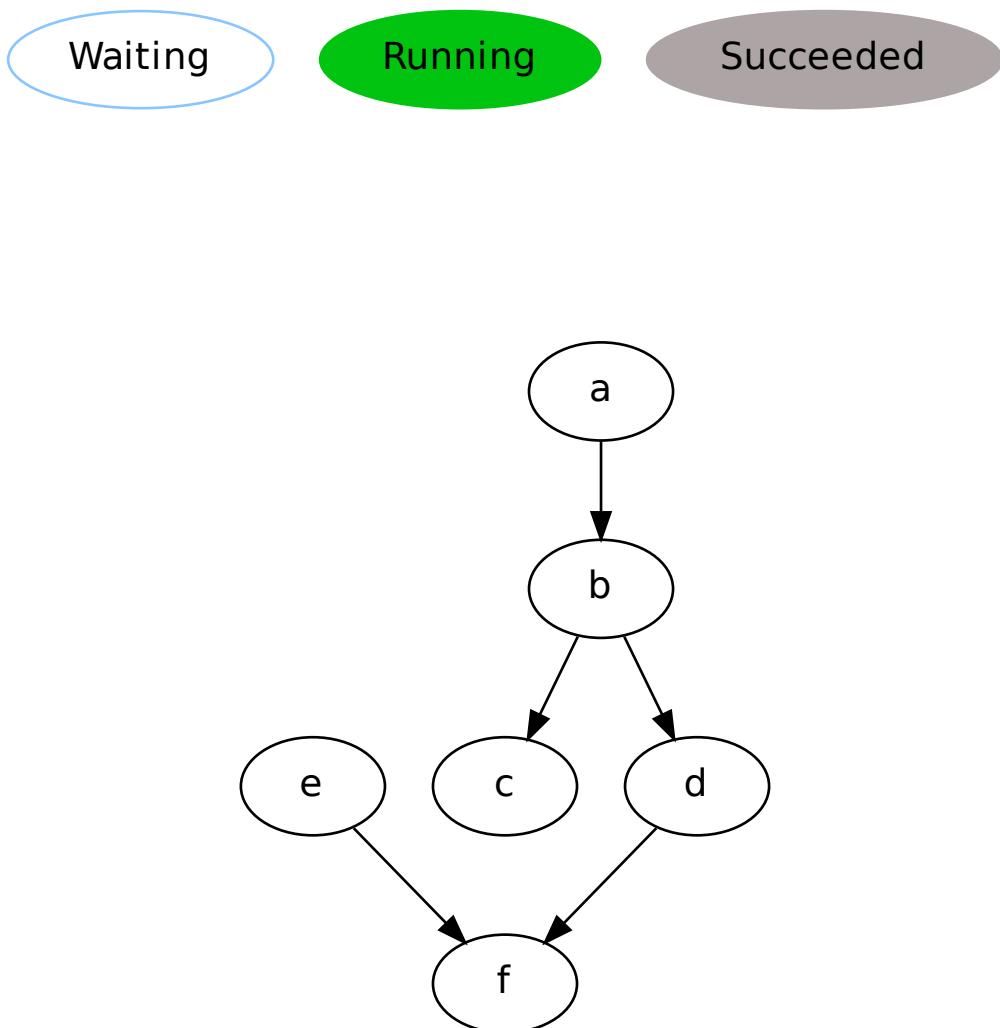
**Succeeded** If the *job* submitted by a *task* has successfully completed (i.e. there is zero return code) then it is said to have succeeded.

These descriptions, and a few more (e.g. failed), are called the *task states*.

## The Cycl GUI

To help you to keep track of a running suite CycL has a graphical user interface (the CycL GUI) which can be used for monitoring and interaction.

The CycL GUI looks quite like `cyclc graph` but the tasks are colour-coded to represent their state, as in the following diagram.



This is the “graph view”. The CycL GUI has two other views called “tree” and “dot”.

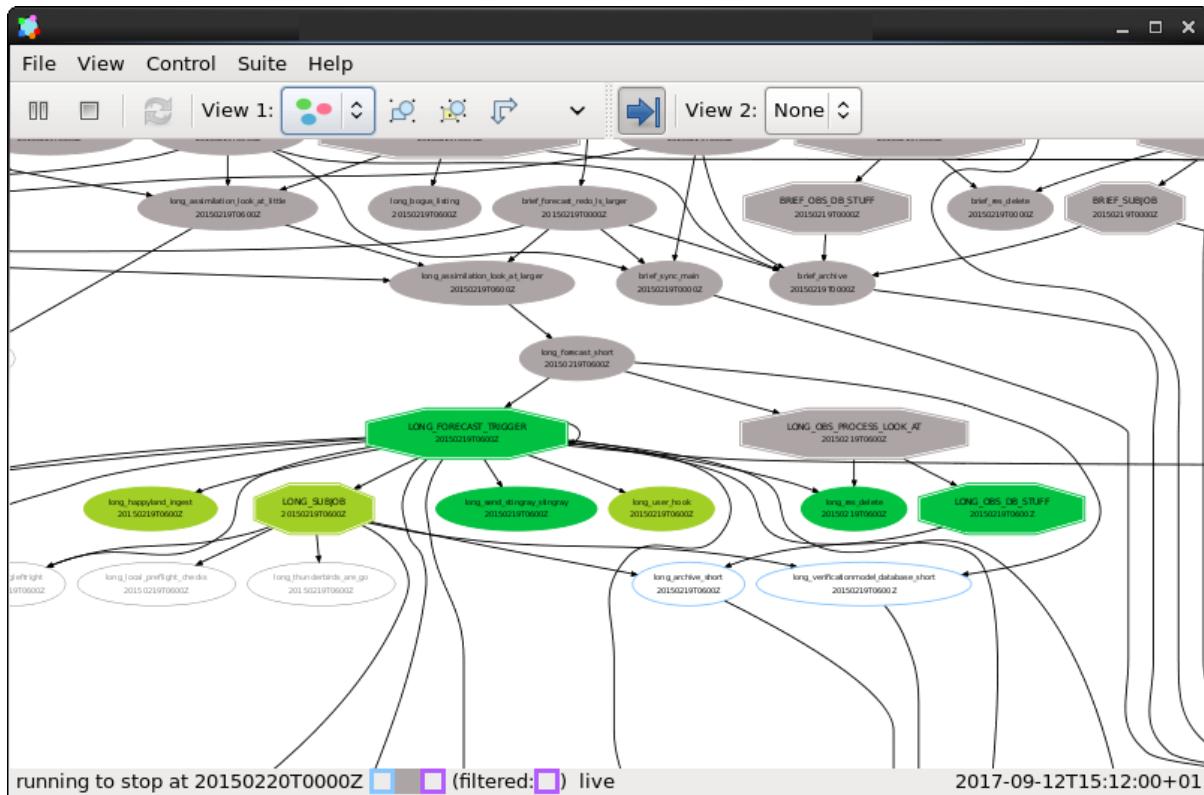


Fig. 1: Screenshot of the Cyc GUI in “Graph View” mode.

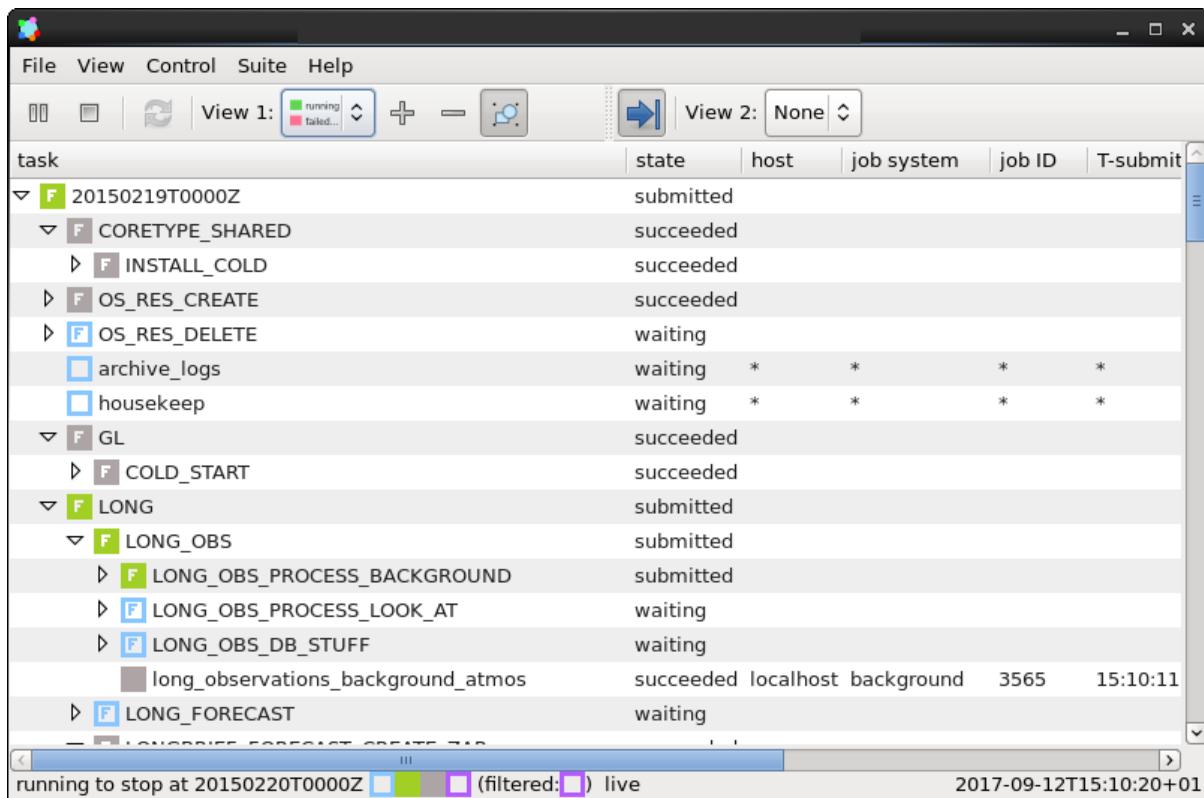


Fig. 2: Screenshot of the Cyc GUI in “Tree View” mode.

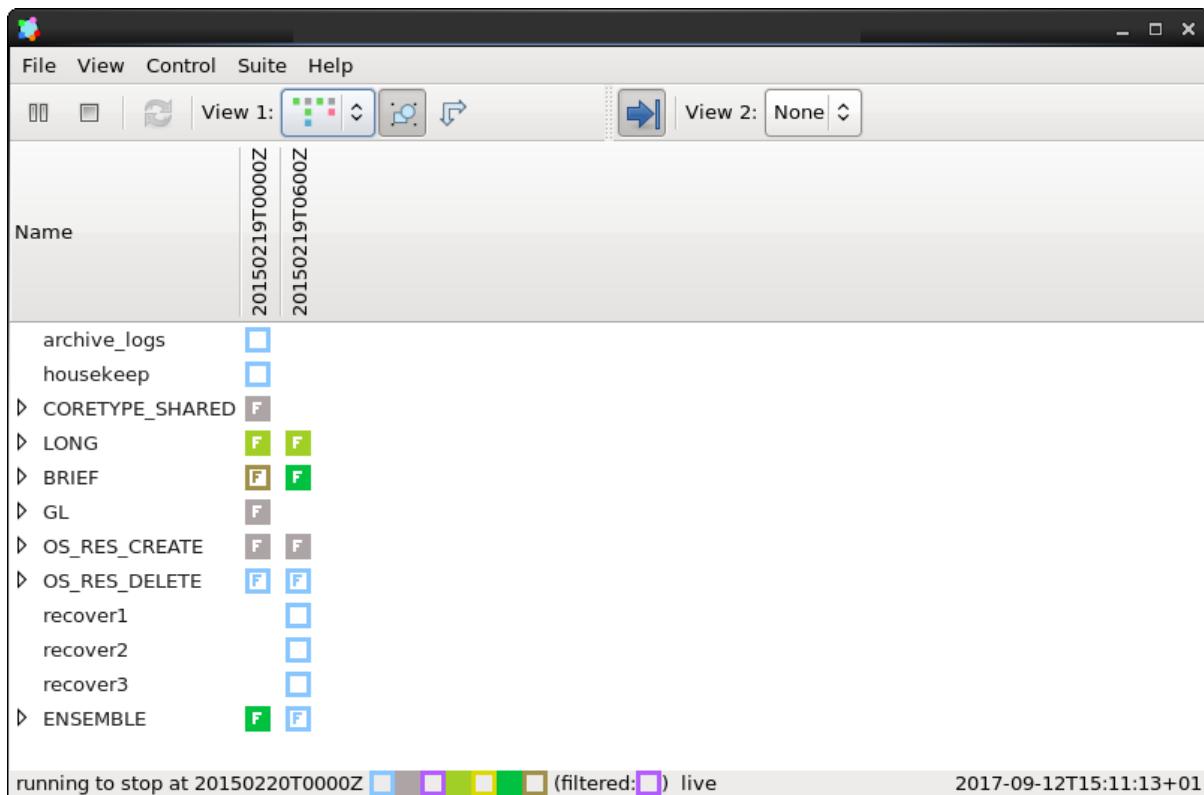


Fig. 3: Screenshot of the Cyc GUI in “Dot View” mode.

## Where Do All The Files Go?

### The Work Directory

When a *task* is run Cyc creates a directory for the *job* to run in. This is called the *work directory*.

By default the work directory is located in a directory structure under the relevant *cycle point* and *task* name:

```
~/cylc-run/<suite-name>/work/<cycle-point>/<task-name>
```

### The Job Log Directory

When a task is run Cyc generates a *job script* which is stored in the *job log directory* as the file *job*.

When the *job script* is executed the stdout and stderr are redirected into the *job.out* and *job.err* files which are also stored in the *job log directory*.

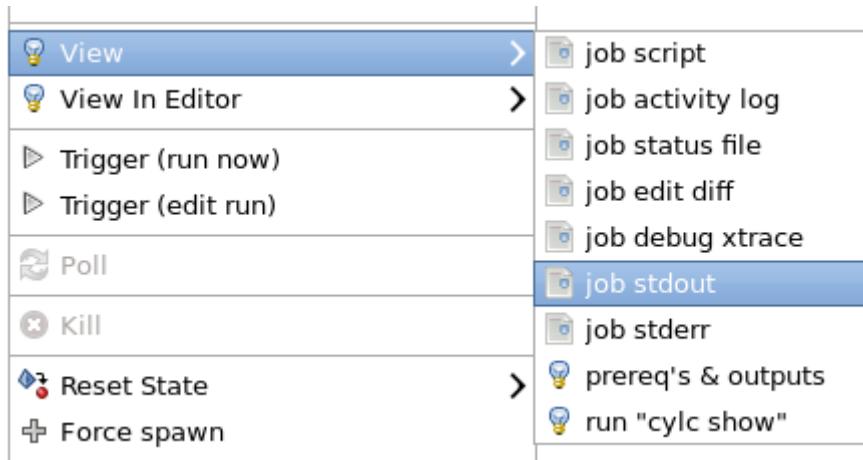
The *job log directory* lives in a directory structure under the *cycle point*, *task name* and *job submission number*:

```
~/cylc-run/<suite-name>/log/job/<cycle-point>/<task-name>/<job-submission-num>/
```

The *job submission number* starts at 1 and increments by 1 each time a task is re-run.

---

**Tip:** If a task has run and is still visible in the Cyc GUI you can view its *job log files* by right-clicking on the task and selecting “View”.



---

## Running A Suite

It is a good idea to check a suite for errors before running it. Cyc provides a command which automatically checks for any obvious configuration issues called `cylc validate`, run via:

```
cylc validate <path/to/suite>
```

Here `<path/to/suite>` is the path to the suite's location within the filesystem (so if we create a suite in `~/cylc-run/foo` we would put `~/cylc-run/foo/suite.rc`).

Next we can run the suite using the `cylc run` command.

```
cylc run <name>
```

The name is the name of the *suite directory* (i.e. `<name>` would be `foo` in the above example).

---

**Note:** In this tutorial we are writing our suites in the `cylc-run` directory.

It is possible to write them elsewhere on the system. If we do so we must register the suite with Cyc before use.

We do this using the `cylc reg` command which we supply with a name which will be used to refer to the suite in place of the path i.e:

```
cylc reg <name> <path/to/suite>
cylc validate <name>
cylc run <name>
```

The `cylc reg` command will create a directory for the suite in the `cylc-run` directory meaning that we will have separate *suite directories* and *run directories*.

---

## Suite Files

Cyc generates files and directories when it runs a suite, namely:

**log/** Directory containing log files, including:

**log/db** The database which Cyc uses to record the state of the suite;

**log/job** The directory where the *job log files* live;

**log/suite** The directory where the *suite log files* live. These files are written by Cyc as the suite is run and are useful for debugging purposes in the event of error.

**suite.rc.processed** A copy of the suite.rc file made after any [Jinja2](http://jinja.pocoo.org/) (<http://jinja.pocoo.org/>) has been processed - we will cover this in the *Consolidating Configuration* (page 53) section.

**share/** The *share directory* is a place where *tasks* can write files which are intended to be shared within that cycle.

**work/** A directory hierarchy containing task's *work directories*.

## Practical

In this practical we will add some scripts to, and run, the weather forecasting suite from the scheduling tutorial.

### 1. Create A New Suite.

The following command will copy some files for us to work with into a new suite called runtime-introduction:

```
rose tutorial runtime-introduction
cd ~/cylc-run/runtime-introduction
```

In this directory we have the suite.rc file from the *weather forecasting suite* (page 34) with some runtime configuration added to it.

There is also a script called get-observations located in the bin directory.

Take a look at the [runtime] section in the suite.rc file.

### 2. Run The Suite.

First validate the suite by running:

```
cylc validate .
```

Open the Cylc GUI (in the background) by running the following command:

```
cylc gui runtime-introduction &
```

Finally run the suite by executing:

```
cylc run runtime-introduction
```

The tasks will start to run - you should see them going through the “Waiting”, “Running” and “Succeeded” states.

When the suite reaches the final cycle point and all tasks have succeeded it will shutdown automatically and the GUI will go blank.

**Tip:** You can also run a suite from the Cylc GUI by pressing the “play” button.



A box will appear. Ensure that “Cold Start” is selected then press “Start”.



### 3. Inspect A Job Log.

Try opening the file `job.out` for one of the `get_observations` jobs in a text editor. The file will be located within the *job log directory*:

```
log/job/<cycle-point>/get_observations_heathrow/01/job.out
```

You should see something like this:

```
Suite      : runtime-introduction
Task Job  : 20000101T0000Z/get_observations_heathrow/01 (try 1)
User@Host : username@hostname

Guessing Weather Conditions
Writing Out Wind Data
1970-01-01T00:00:00Z NORMAL - started
2038-01-19T03:14:08Z NORMAL - succeeded
```

- The first three lines are information which Cyc has written to the file to provide information about the job.
- The last two lines were also written by cyc. They provide timestamps marking the stages in the job's life.
- The lines in the middle are the stdout of the job itself.

### 4. Inspect A Work Directory.

The `get_rainfall` task should create a file called `rainfall` in its *work directory*. Try opening this file, recalling that the format of the relevant path from within the work directory will be:

```
work/<cycle-point>/get_rainfall/rainfall
```

**Hint:** The `get_rainfall` task only runs every third cycle.

### 5. Extension: Explore The Cyc GUI

- Try re-running the suite.
- Try changing the current view(s).

**Tip:** You can do this from the “View” menu or from the toolbar:



- Try pressing the “Pause” button which is found in the top left-hand corner of the GUI.
- Try right-clicking on a task. From the right-click menu you could try:
  - “Trigger (run now)”
  - “Reset State”

### 3.3.2 Runtime Configuration

In the last section we associated tasks with scripts and ran a simple suite. In this section we will look at how we can configure these tasks.

#### Environment Variables

We can specify environment variables in a task’s [environment] section. These environment variables are then provided to *jobs* when they run.

```
[runtime]
  [[countdown]]
    script = seq $START_NUMBER
  [[[environment]]]
    START_NUMBER = 5
```

Each job is also provided with some standard environment variables e.g:

**CYLC\_SUITE\_RUN\_DIR** The path to the suite’s *run directory* (e.g. `~/cylc-run/suite`).

**CYLC\_TASK\_WORK\_DIR** The path to the associated task’s *work directory* (e.g. `run-directory/work/cycle/task`).

**CYLC\_TASK\_CYCLE\_POINT** The *cycle point* for the associated task (e.g. `20171009T0950`).

There are many more environment variables - see the **Cyclc User Guide** (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>) for more information.

#### Job Submission

By default Cyclc runs *jobs* on the machine where the suite is running. We can tell Cyclc to run jobs on other machines by setting the [remote]host setting to the name of the host, e.g. to run a task on the host `computehost` you might write:

```
[runtime]
  [[hello_computehost]]
    script = echo "Hello Compute Host"
  [[[remote]]]
    host = computehost
```

Cyclc also executes jobs as *background processes* ([https://en.wikipedia.org/wiki/Background\\_process](https://en.wikipedia.org/wiki/Background_process)) by default. When we are running jobs on other compute hosts we will often want to use a *batch system* (job scheduler) ([https://en.wikipedia.org/wiki/Job\\_scheduler](https://en.wikipedia.org/wiki/Job_scheduler)) to submit our job. Cyclc supports the following *batch systems*:

- at
- loadleveler
- lsf
- pbs

- sge
- slurm
- moab

*Batch systems* typically require *directives* in some form. *Directives* inform the *batch system* of the requirements of a *job*, for example how much memory a given job requires or how many CPUs the job will run on. For example:

```
[runtime]
  [[big_task]]
    script = big-executable

    # Submit to the host "big-computer".
    [[[remote]]]
      host = big-computer

    # Submit the job using the "slurm" batch system.
    [[[job]]]
      batch system = slurm

    # Inform "slurm" that this job requires 500MB of RAM and 4 CPUs.
    [[[directives]]]
      --mem = 500
      --ntasks = 4
```

## Timeouts

We can specify a time limit after which a job will be terminated using the [job] execution time limit setting. The value of the setting is an *ISO8601 duration*. Cyc automatically inserts this into a job's directives as appropriate.

```
[runtime]
  [[some_task]]
    script = some-executable
  [[[job]]]
    execution time limit = PT15M # 15 minutes.
```

## Retries

Sometimes jobs fail. This can be caused by two factors:

- Something going wrong with the job's execution e.g:
  - A bug;
  - A system error;
  - The job hitting the execution time limit.
- Something going wrong with the job submission e.g:
  - A network problem;
  - The *job host* becoming unavailable or overloaded;
  - An issue with the directives.

In the event of failure Cyc can automatically re-submit (retry) jobs. We configure retries using the [job] execution retry delays and [job] submission retry delays settings. These settings are both set to an *ISO8601 duration*, e.g. setting execution retry delays to PT10M would cause the job to retry every 10 minutes in the event of execution failure.

We can limit the number of retries by writing a multiple in front of the duration, e.g:

```
[runtime]
  [[some-task]]
    script = some-script
  [[[job]]]
    # In the event of execution failure, retry a maximum
    # of three times every 15 minutes.
    execution retry delays = 3*PT15M
    # In the event of submission failure, retry a maximum
    # of two times every ten minutes and then every 30
    # minutes thereafter.
    submission retry delays = 2*PT10M, PT30M
```

## Start, Stop, Restart

We have seen how to start and stop Cycl suites with `cyclc run` and `cyclc stop` respectively. The `cyclc stop` command causes Cycl to wait for all running jobs to finish before it stops the suite. There are two options which change this behaviour:

**`cyclc stop --kill`** When the `--kill` option is used Cycl will kill all running jobs before stopping. *Cyclc can kill jobs on remote hosts and uses the appropriate command when a batch system is used.*

**`cyclc stop --now --now`** When the `--now` option is used twice Cycl stops straight away, leaving any jobs running.

Once a suite has stopped it is possible to restart it using the `cyclc restart` command. When the suite restarts it picks up where it left off and carries on as normal.

```
# Run the suite "name".
cyclc run <name>
# Stop the suite "name", killing any running tasks.
cyclc stop <name> --kill
# Restart the suite "name", picking up where it left off.
cyclc restart <name>
```

---

## Practical

**In this practical we will add runtime configuration to the weather-forecasting suite from the scheduling tutorial.**

### 1. Create A New Suite.

Create a new suite by running the command:

```
rose tutorial runtime-tutorial
cd ~/cyclc-run/runtime-tutorial
```

You will now have a copy of the weather-forecasting suite along with some executables and python modules.

### 1. Set The Initial And Final Cycle Points.

First we will set the initial and final cycle points (see the [datetime tutorial](#) (page 31) for help with writing ISO8601 datetimes):

- The *final cycle point* should be set to the time one hour ago from the present time (with minutes and seconds ignored), e.g. if the current time is 9:45 UTC then the final cycle point should be at 8:00 UTC.
- The *initial cycle point* should be the final cycle point minus six hours.

---

## Reminder

Remember that we are working in UTC mode (the +00 time zone). Datetimes should end with a Z character to reflect this.

---

---

### Solution

You can check your answers by running the following commands (hyphens and colons optional but can't be mixed):

**For the initial cycle point:** rose date --utc --offset -PT7H --format  
CCYY-MM-DDThh:00Z

**For the final cycle point:** rose date --utc --offset -PT1H --format  
CCYY-MM-DDThh:00Z

---

Run cylc validate to check for any errors:

```
cylc validate .
```

## 2. Add Runtime Configuration For The get\_observations Tasks.

In the bin directory is a script called get-observations. This script gets weather data from the MetOffice DataPoint (<https://www.metoffice.gov.uk/datapoint>) service. It requires two environment variables:

**SITE\_ID:** A four digit numerical code which is used to identify a weather station, e.g. 3772 is Heathrow Airport.

**API\_KEY:** An authentication key required for access to the service.

Generate a Datapoint API key:

```
rose tutorial api-key
```

Add the following lines to the bottom of the suite.rc file replacing xxx... with your API key:

```
[runtime]
  [[get_observations_heathrow]]
    script = get-observations
  [[[environment]]]
    SITE_ID = 3772
    API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

Add three more get\_observations tasks for each of the remaining weather stations.

You will need the codes for the other three weather stations, which are:

- Camborne - 3808
- Shetland - 3005
- Belmullet - 3976

---

### Solution

```
[runtime]
  [[get_observations_heathrow]]
    script = get-observations
  [[[environment]]]
    SITE_ID = 3772
    API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
  [[get_observations_camborne]]
    script = get-observations
```

(continues on next page)

(continued from previous page)

```

[[[environment]]]
SITE_ID = 3808
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

[[get_observations_shetland]]
script = get-observations
[[[environment]]]
SITE_ID = 3005
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

[[get_observations_belmullet]]
script = get-observations
[[[environment]]]
SITE_ID = 3976
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

```

Check the suite.rc file is valid by running the command:

```
cylc validate .
```

### 3. Test The get\_observations Tasks.

Next we will test the get\_observations tasks.

Open the Cylc GUI by running the following command:

```
cylc gui runtime-tutorial &
```

Run the suite either by pressing the play button in the Cylc GUI or by running the command:

```
cylc run runtime-tutorial
```

If all goes well the suite will startup and the tasks will run and succeed. Note that the tasks which do not have a [runtime] section will still run though they will not do anything as they do not call any scripts.

Once the suite has reached the final cycle point and all tasks have succeeded the suite will automatically shutdown.

The get-observations script produces a file called wind.csv which specifies the wind speed and direction. This file is written in the task's *work directory*.

Try and open one of the wind.csv files. Note that the path to the *work directory* is:

```
work/<cycle-point>/<task-name>
```

You should find a file containing four numbers:

- The longitude of the weather station;
- The latitude of the weather station;
- The wind direction (*the direction the wind is blowing towards*) in degrees;
- The wind speed in miles per hour.

---

#### Hint

If you run ls work you should see a list of cycles. Pick one of them and open the file:

```
work/<cycle-point>/get_observations_heathrow/wind.csv
```

---

### 4. Add runtime configuration for the other tasks.

The runtime configuration for the remaining tasks has been written out for you in the `runtime` file which you will find in the *suite directory*. Copy the code in the `runtime` file to the bottom of the `suite.rc` file.

Check the `suite.rc` file is valid by running the command:

```
cylc validate .
```

### 5. Run The Suite.

Open the Cycl GUI (if not already open) and run the suite.

---

#### Hint

```
cylc gui runtime-tutorial &
```

Run the suite either by:

- Pressing the play button in the Cycl GUI. Then, ensuring that “Cold Start” is selected within the dialogue window, pressing the “Start” button.
  - Running the command `cylc run runtime-tutorial`.
- 

### 6. View The Forecast Summary.

The `post_process_exeter` task will produce a one-line summary of the weather in Exeter, as forecast two hours ahead of time. This summary can be found in the `summary.txt` file in the *work directory*.

Try opening the summary file - it will be in the last cycle. The path to the *work directory* is:

```
work/<cycle-point>/<task-name>
```

---

#### Hint

- `cycle-point` - this will be the last cycle of the suite, i.e. the final cycle point.
  - `task-name` - set this to “`post_process_exeter`”.
- 

### 7. View The Rainfall Data.

The `forecast` task will produce a html page where the rainfall data is rendered on a map. This html file is called `job-map.html` and is saved alongside the *job log*.

Try opening this file in a web browser, e.g via:

```
firefox <filename> &
```

The path to the *job log directory* is:

```
log/job/<cycle-point>/<task-name>/<submission-number>
```

---

#### Hint

- `cycle-point` - this will be the last cycle of the suite, i.e. the final cycle point.
  - `task-name` - set this to “`forecast`”.
  - `submission-number` - set this to “01”.
-

### 3.3.3 Consolidating Configuration

In the last section we wrote out the following code in the `suite.rc` file:

```
[runtime]
  [[get_observations_heathrow]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3772
      API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
  [[get_observations_camborne]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3808
      API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
  [[get_observations_shetland]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3005
      API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
  [[get_observations_belmullet]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3976
      API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

In this code the `script` item and the `API_KEY` environment variable have been repeated for each task. This is bad practice as it makes the configuration lengthy and making changes can become difficult.

Likewise the graphing relating to the `get_observations` tasks is highly repetitive:

```
[scheduling]
  [[dependencies]]
    [[[T00/PT3H]]]
      graph =
        """
          get_observations_belmullet => consolidate_observations
          get_observations_camborne => consolidate_observations
          get_observations_heathrow => consolidate_observations
          get_observations_shetland => consolidate_observations
        """
```

Cyc offers three ways of consolidating configurations to help improve the structure of a suite and avoid duplication.

#### Families

*Families* provide a way of grouping tasks together so they can be treated as one.

#### Runtime

*Families* are groups of tasks which share a common configuration. In the present example the common configuration is:

```
script = get-observations
[[[environment]]]
  API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

We define a family as a new task consisting of the common configuration. By convention families are named in upper case:

```
[[GET_OBSERVATIONS]]
script = get-observations
[[[environment]]]
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

We “add” tasks to a family using the `inherit` setting:

```
[[get_observations_heathrow]]
inherit = GET_OBSERVATIONS
[[[environment]]]
SITE_ID = 3772
```

When we add a task to a family in this way it *inherits* the configuration from the family, i.e. the above example is equivalent to:

```
[[get_observations_heathrow]]
script = get-observations
[[[environment]]]
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
SITE_ID = 3772
```

It is possible to override inherited configuration within the task. For example if we wanted the `get_observations_heathrow` task to use a different API key we could write:

```
[[get_observations_heathrow]]
inherit = GET_OBSERVATIONS
[[[environment]]]
API_KEY = special-api-key
SITE_ID = 3772
```

Using families the `get_observations` tasks could be written like so:

```
[runtime]
[[GET_OBSERVATIONS]]
script = get-observations
[[[environment]]]
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

[[get_observations_heathrow]]
inherit = GET_OBSERVATIONS
[[[environment]]]
SITE_ID = 3772
[[get_observations_camborne]]
inherit = GET_OBSERVATIONS
[[[environment]]]
SITE_ID = 3808
[[get_observations_shetland]]
inherit = GET_OBSERVATIONS
[[[environment]]]
SITE_ID = 3005
[[get_observations_belmullet]]
inherit = GET_OBSERVATIONS
[[[environment]]]
SITE_ID = 3976
```

## Graphing

*Families* can be used in the suite’s *graph*, e.g:

```
GET_OBSERVATIONS:succeed_all => consolidate_observations
```

The `:succeed-all` is a special *qualifier* which in this example means that the `consolidate_observations` task will run once *all* of the members of the `GET_OBSERVATIONS` family have succeeded. This is equivalent to:

```
get_observations_heathrow => consolidate_observations
get_observations_camborne => consolidate_observations
get_observations_shetland => consolidate_observations
get_observations_belmullet => consolidate_observations
```

The `GET_OBSERVATIONS:succeed-all` part is referred to as a *family trigger*. Family triggers use special qualifiers which are non-optional. The most commonly used ones are:

**succeed-all** Run if all of the members of the family have succeeded.

**succeed-any** Run as soon as any one family member has succeeded.

**finish-all** Run as soon as all of the family members have completed (i.e. have each either succeeded or failed).

For more information on family triggers see the Cylc User Guide (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

## The root Family

There is a special family called *root* (in lowercase) which is used only in the runtime to provide configuration which will be inherited by all tasks.

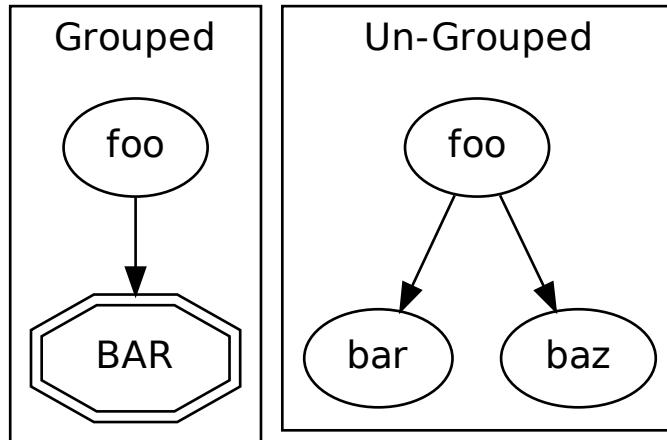
In the following example the task `bar` will inherit the environment variable `FOO` from the `[root]` section:

```
[runtime]
  [[root]]
    [[[environment]]]
      FOO = foo
  [[bar]]
    script = echo $FOO
```

## Families and cylc graph

By default, `cylc graph` groups together all members of a family in the *graph*. To un-group a family right click on it and select *UnGroup*.

For instance if the tasks `bar` and `baz` both inherited from `BAR` `cylc graph` would produce:



---

## Practical

In this practical we will consolidate the configuration of the weather-forecasting suite from the previous section.

### 1. Create A New Suite.

To make a new copy of the forecasting suite run the following commands:

```
rose tutorial consolidation-tutorial  
cd ~/cylc-run/consolidation-tutorial
```

Set the intial and final cycle points as you did in the *previous tutorial* (page 49).

### 2. Move Site-Wide Settings Into The root Family.

The following three environment variables are used by multiple tasks:

```
PYTHONPATH="$CYLC_SUITE_DEF_PATH/lib/python:$PYTHONPATH"  
RESOLUTION = 0.2  
DOMAIN = -12,48,5,61 # Do not change!
```

Rather than manually adding them to each task individually we could put them in the `root` family, making them accessible to all tasks.

Add a `root` section containing these three environment variables. Remove the variables from any other task's environment sections:

```
[runtime]  
+   [[root]]  
+     [[[environment]]]  
+       # Add the `python` directory to the PYTHONPATH.  
+       PYTHONPATH="$CYLC_SUITE_DEF_PATH/lib/python:$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!
```

```

[[consolidate_observations]]
script = consolidate-observations
[[[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!

[[get_rainfall]]
script = get-rainfall
[[[environment]]]
# The key required to get weather data from the DataPoint service.
# To use archived data comment this line out.
API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
# 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!

[[forecast]]
script = forecast 60 5 # Generate 5 forecasts at 60 minute intervals.
[[[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!
# The path to the files containing wind data (the {variables} will
# get substituted in the forecast script).
WIND_FILE_TEMPLATE = $CYLC_SUITE_WORK_DIR/{cycle}/consolidate_
observations/wind_{xy}.csv
# List of cycle points to process wind data from.
WIND_CYCLES = 0, -3, -6

# The path to the rainfall file.
RAINFALL_FILE = $CYLC_SUITE_WORK_DIR/$CYLC_TASK_CYCLE_POINT/get_
rainfall/rainfall.csv
# Create the html map file in the task's log directory.
MAP_FILE = "${CYLC_TASK_LOG_ROOT}-map.html"
# The path to the template file used to generate the html map.
MAP_TEMPLATE = "$CYLC_SUITE_RUN_DIR/lib/template/map.html"

[[post_process_exeter]]
# Generate a forecast for Exeter 60 minutes into the future.
script = post-process exeter 60
[[[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!

```

To ensure that the environment variables are being inherited correctly by the tasks, inspect the [runtime] section using `cylc get-config` by running the following command:

```
cylc get-config . --sparse -i "[runtime]"
```

You should see the environment variables from the [root] section in the [environment] section for all tasks.

---

**Tip:** You may find it easier to open the output of this command in a text editor, e.g:

```
cylc get-config . --sparse -i "[runtime]" | gvim -
```

---

## Jinja2

*Jinja2* (page 58) is a templating language often used in web design with some similarities to python. It can be used to make a suite definition more dynamic.

### The Jinja2 Language

In Jinja2 statements are wrapped with { % characters, i.e:

```
{% ... %}
```

Variables are initiated using the set statement, e.g:

```
{% set foo = 3 %}
```

Expressions wrapped with { { characters will be replaced with the value of the evaluation of the expression, e.g:

```
There are {{ foo }} methods for consolidating the suite.rc file
```

Would result in:

```
There are 3 methods for consolidating the suite.rc file
```

Loops are written with for statements, e.g:

```
{% for x in range(foo) %}
{{ x }}
{% endfor %}
```

Would result in:

```
0
1
2
```

To enable Jinja2 in the suite.rc file, add the following shebang ([https://en.wikipedia.org/wiki/Shebang\\_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))) to the top of the file:

```
#!/usr/bin/python
```

For more information see the [Jinja2 Tutorial](http://jinja.pocoo.org/docs) (<http://jinja.pocoo.org/docs>).

### Example

To consolidate the configuration for the get\_observations tasks we could define a dictionary of station and ID pairs:

```
{% set stations = {'belmullet': 3976,
                  'camborne': 3808,
                  'heathrow': 3772,
                  'shetland': 3005} %}
```

We could then loop over the stations like so:

```
{% for station in stations %}
  {{ station }}
{% endfor %}
```

After processing, this would result in:

```
belmullet
camborne
heathrow
shetland
```

We could also loop over both the stations and corresponding IDs like so:

```
{% for station, id in stations.items() %}
  {{ station }} - {{ id }}
{% endfor %}
```

This would result in:

```
belmullet - 3976
camborne - 3808
heathrow - 3772
shetland - 3005
```

Putting this all together, the `get_observations` configuration could be written as follows:

```
#!Jinja2

{% set stations = {'belmullet': 3976,
                  'camborne': 3808,
                  'heathrow': 3772,
                  'shetland': 3005} %}

[scheduling]
  [[dependencies]]
    [[[T00/PT3H]]]
      graph = """
{% for station in stations %}
  get_observations_{{station}} => consolidate_observations
{% endfor %}
"""

[runtime]
{% for station, id in stations.items() %}
  [[get_observations_{{station}}]]
    script = get-observations
    [[[environment]]]
      SITE_ID = {{ id }}
      API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
    {{ endfor }}
```

---

## Practical

This practical continues on from the families practical.

### 3. Use Jinja2 To Avoid Duplication.

The API\_KEY environment variable is used by both the get\_observations and get\_rainfall tasks. Rather than writing it out multiple times we will use Jinja2 to centralise this configuration.

At the top of the suite.rc file add the Jinja2 shebang line. Then copy the value of the API\_KEY environment variable and use it to define an API\_KEY Jinja2 variable:

```
#!Jinja2

{% set API_KEY = 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' %}
```

Next replace the key, where it appears in the suite, with {{ API\_KEY }}:

```
[runtime]
  [[get_observations_heathrow]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3772
-       API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
+       API_KEY = {{ API_KEY }}
  [[get_observations_camborne]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3808
-       API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
+       API_KEY = {{ API_KEY }}
  [[get_observations_shetland]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3005
-       API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
+       API_KEY = {{ API_KEY }}
  [[get_observations_belmullet]]
    script = get-observations
    [[[environment]]]
      SITE_ID = 3976
-       API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
+       API_KEY = {{ API_KEY }}
  [[get_rainfall]]
    script = get-rainfall
    [[[environment]]]
      # The key required to get weather data from the DataPoint service.
      # To use archived data comment this line out.
-       API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
+       API_KEY = {{ API_KEY }}
```

Check the result with cylc get-config. The Jinja2 will be processed so you should not see any difference after making these changes.

---

## Parameterised Tasks

Parameterised tasks (see *parameterisation*) provide a way of implicitly looping over tasks without the need for Jinja2.

### Cylc Parameters

Parameters are defined in their own section, e.g:

```
[cylc]
  [[parameters]]
    world = Mercury, Venus, Earth
```

They can then be referenced by writing the name of the parameter in angle brackets, e.g:

```
[scheduling]
  [[dependencies]]
    graph = start => hello<world> => end
[runtime]
  [[hello<world>]]
    script = echo 'Hello World!'
```

When the `suite.rc` file is read by Cylc, the parameters will be expanded. For example the code above is equivalent to:

```
[scheduling]
  [[dependencies]]
    graph = """
      start => hello_Mercury => end
      start => hello_Venus => end
      start => hello_Earth => end
    """
[runtime]
  [[hello_Mercury]]
    script = echo 'Hello World!'
  [[hello_Venus]]
    script = echo 'Hello World!'
  [[hello_Earth]]
    script = echo 'Hello World!'
```

We can refer to a specific parameter by writing it after an `=` sign:

```
[runtime]
  [[hello<world=Earth>]]
    script = echo 'Greetings Earth!'
```

## Environment Variables

The name of the parameter is provided to the job as an environment variable called `CYLC_TASK_PARAM_<parameter>` where `<parameter>` is the name of the parameter (in the present case `world`):

```
[runtime]
  [[hello<world>]]
    script = echo "Hello ${CYLC_TASK_PARAM_world}!"
```

## Parameter Types

Parameters can be either words or integers:

```
[cylc]
  [[parameters]]
    foo = 1..5
    bar = 1..5..2
    baz = pub, qux, bol
```

---

**Hint:** Remember that CycL automatically inserts an underscore between the task and the parameter, e.g. the following lines are equivalent:

```
task<baz=pub>
task_pub
```

---

---

**Hint:** When using integer parameters, to prevent confusion, CycL prefixes the parameter value with the parameter name. For example:

```
[scheduling]
  [[dependencies]]
    graph = """
      # task<bar> would result in:
      task_bar1
      task_bar3
      task_bar5

      # task<baz> would result in:
      task_pub
      task_qux
      task_bol
    """
```

---

Using parameters the get\_observations configuration could be written like so:

```
[scheduling]
  [[dependencies]]
    [[[T00/PT3H]]]
    graph = """
      get_observations<station> => consolidate_observations
    """

[runtime]
  [[get_observations<station>]]
    script = get-observations
    [[[environment]]]
      API_KEY = xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

  [[get_observations<station=belmullet>]]
    [[[environment]]]
      SITE_ID = 3976
  [[get_observations<station=camborne>]]
    [[[environment]]]
      SITE_ID = 3808
  [[get_observations<station=heathrow>]]
    [[[environment]]]
      SITE_ID = 3772
  [[get_observations<station=shetland>]]
    [[[environment]]]
      SITE_ID = 3005
```

---

For more information see the [Cylc User Guide](http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide) (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

---

## Practical

This practical continues on from the [Jinja2 practical](#).

#### 4. Use Parameterisation To Consolidate The get\_observations Tasks.

Next we will parameterise the get\_observations tasks.

Add a parameter called station:

```
[cylc]
    UTC mode = True
+   [[parameters]]
+     station = belmullet, camborne, heathrow, shetland
```

Remove the four get\_observations tasks and insert the following code in their place:

```
[[get_observations<station>]]
  script = get-observations
  [[[environment]]]
    API_KEY = {{ API_KEY }}
```

Using cylc get-config you should see that Cycl replaces the <station> with each of the stations in turn, creating a new task for each:

```
cylc get-config . --sparse -i "[runtime]"
```

The get\_observations tasks are now missing the SITE\_ID environment variable. Add a new section for each station with a SITE\_ID:

```
[[get_observations<station=heathrow>]]
  [[[environment]]]
    SITE_ID = 3772
```

**Hint:** The relevant IDs are:

- Belmullet - 3976
- Camborne - 3808
- Heathrow - 3772
- Shetland - 3005

#### Solution

```
[[get_observations<station=belmullet>]]
  [[[environment]]]
    SITE_ID = 3976
[[get_observations<station=camborne>]]
  [[[environment]]]
    SITE_ID = 3808
[[get_observations<station=heathrow>]]
  [[[environment]]]
    SITE_ID = 3772
[[get_observations<station=shetland>]]
  [[[environment]]]
    SITE_ID = 3005
```

Using cylc get-config you should now see four get\_observations tasks, each with a script, an API\_KEY and a SITE\_ID:

```
cylc get-config . --sparse -i "[runtime]"
```

Finally we can use this parameterisation to simplify the suite's graphing. Replace the `get_observations` lines in the graph with `get_observations<station>`:

```
[[[PT3H]]]
# Repeat every three hours starting at the initial cycle point.
graph = """
-    get_observations_belmullet => consolidate_observations
-    get_observations_camborne => consolidate_observations
-    get_observations_heathrow => consolidate_observations
-    get_observations_shetland => consolidate_observations
+    get_observations<station> => consolidate_observations
"""

```

---

**Hint:** The `cylc get-config` command does not expand parameters or families in the graph so you must use `cylc graph` to inspect changes to the graphing.

---

## 5. Use Parameterisation To Consolidate The `post_process` Tasks.

At the moment we only have one `post_process` task (`post_process_exeter`), but suppose we wanted to add a second task for Edinburgh.

Create a new parameter called `site` and set it to contain `exeter` and `edinburgh`. Parameterise the `post_process` task using this parameter.

---

**Hint:** The first argument to the `post-process` task is the name of the site. We can use the `CYLC_TASK_PARAM_site` environment variable to avoid having to write out this section twice.

---

### Solution

First we must create the `site` parameter:

```
[cylc]
UTC mode = True
[[parameters]]
    station = belmullet, camborne, heathrow, shetland
+    site = exeter, edinburgh
```

Next we parameterise the task in the graph:

```
-get_rainfall => forecast => post_process_exeter
+get_rainfall => forecast => post_process<site>
```

And also the runtime:

```
-[[post_process_exeter]]
+[[post_process<site>]]
    # Generate a forecast for Exeter 60 minutes in the future.
-    script = post-process exeter 60
+    script = post-process $CYLC_TASK_PARAM_site 60
```

---

## The `cylc get-config` Command

The `cylc get-config` command reads in then prints out the `suite.rc` file to the terminal.

Throughout this section we will be introducing methods for consolidating the `suite.rc` file, the `cylc get-config` command can be used to “expand” the `suite.rc` file back to its full form.

---

**Note:** The main use of `cylc get-config` is inspecting the `[runtime]` section of a suite. The `cylc get-config` command does not expand `parameterisations` and `families` in the suite’s `graph`. To inspect the graphing use the `cylc graph` command.

---

Call `cylc get-config` with the path of the suite (. `if you are already in the suite directory) and the --sparse option which hides default values.`

```
cylc get-config <path> --sparse
```

To view the configuration of a particular section or setting refer to it by name using the `-i` option (see *The suite.rc File Format* (page 14) for details), e.g.:

```
# Print the contents of the [scheduling] section.
cylc get-config <path> --sparse -i '[scheduling]'
# Print the contents of the get_observations_heathrow task.
cylc get-config <path> --sparse -i '[runtime][get_observations_heathrow]'
# Print the value of the script setting in the get_observations_heathrow task
cylc get-config <path> --sparse -i '[runtime][get_observations_heathrow]script'
```

## The Three Approaches

The next three sections cover the three consolidation approaches and how we could use them to simplify the suite from the previous tutorial. *Work through them in order!*

- `families` (page 53)
- `jinja2` (page 58)
- `parameters` (page 60)

## Which Approach To Use

Each approach has its uses. Cycl permits mixing approaches, allowing us to use what works best for us. As a rule of thumb:

- *Families* work best consolidating runtime configuration by collecting tasks into broad groups, e.g. groups of tasks which run on a particular machine or groups of tasks belonging to a particular system.
- *Jinja2* (<http://jinja.pocoo.org/>) is good at configuring settings which apply to the entire suite rather than just a single task, as we can define variables then use them throughout the suite.
- *Parameterisation* works best for describing tasks which are very similar but which have subtly different configurations (e.g. different arguments or environment variables).

## 3.4 Further Topics

This section looks at further topics in cylc.

### 3.4.1 Clock Triggered Tasks

In a `datetime cycling` suite the time represented by the `cycle points` bear no relation to the real-world time. Using clock-triggers we can make tasks wait until their cycle point time before running.

Clock-triggering effectively enables us to tether the “cycle time” to the “real world time” which we refer to as the *wall-clock time*.

## Clock Triggering

When clock-triggering tasks we can use different *offsets* (page 32) to the cycle time as follows:

```
clock-trigger = taskname(CYCLE_OFFSET)
```

---

**Note:** Regardless of the offset used, the task still belongs to the cycle from which the offset has been applied.

---

### Example

Our example suite will simulate a clock chiming on the hour.

Within your `~/cylc-run/clock-trigger` directory create a new directory called `clock-trigger`:

```
mkdir ~/cylc-run/clock-trigger
cd ~/cylc-run/clock-trigger
```

Paste the following code into a `suite.rc` file:

```
[cylc]
UTC mode = True # Ignore DST

[scheduling]
    initial cycle point = TODO
    final cycle point = +P1D # Run for one day
    [[dependencies]]
        [[[PT1H]]]
            graph = bell

[runtime]
    [[root]]
        [[[events]]]
            mail events = failed
    [[bell]]
        env-script = eval $(rose task-env)
        script = printf 'bong%.0s\n' $(seq 1 $(cylc cyclepoint --print-hour))
```

Change the initial cycle point to 00:00 this morning (e.g. if it was the first of January 2000 we would write 2000-01-01T00Z).

We now have a simple suite with a single task that prints “bong” a number of times equal to the (cycle point) hour.

Run your suite using:

```
cylc run clock-trigger
```

Stop the suite after a few cycles using the *stop* button in the `cylc` gui. Notice how the tasks run as soon as possible rather than waiting for the actual time to be equal to the cycle point.

## Clock-Triggering Tasks

We want our clock to only ring in real-time rather than the simulated cycle time.

To do this, add the following lines to the `[scheduling]` section of your `suite.rc`:

```
[[special tasks]]
  clock-trigger = bell(PT0M)
```

This tells the suite to clock trigger the `bell` task with a cycle offset of 0 hours.

Save your changes and run your suite.

Your suite should now be running the `bell` task in real-time. Any cycle times that have already passed (such as the one defined by `initial cycle time`) will be run as soon as possible, while those in the future will wait for that time to pass.

At this point you may want to leave your suite running until the next hour has passed in order to confirm the clock triggering is working correctly. Once you are satisfied, stop your suite.

By making the `bell` task a clock triggered task we have made it run in real-time. Thus, when the wall-clock time caught up with the cycle time, the `bell` task triggered.

## Adding More Clock-Triggered Tasks

We will now modify our suite to run tasks at quarter-past, half-past and quarter-to the hour.

Open your `suite.rc` and modify the `[runtime]` section by adding the following:

```
[[quarter_past, half_past, quarter_to]]
  script = echo 'chimes'
```

Edit the `[[scheduling]]` section to read:

```
[[special tasks]]
  clock-trigger = bell(PT0M), quarter_past(PT15M), half_past(PT30M), quarter_
  ↪to(PT45M)
[[dependencies]]
  [[[PT1H]]]
    graph = """
      bell
      quarter_past
      half_past
      quarter_to
    """
```

Note the different values used for the cycle offsets of the clock-trigger tasks.

Save your changes and run your suite using:

```
cylc run clock-trigger now
```

---

**Note:** The `now` argument will run your suite using the current time for the initial cycle point.

---

Again, notice how the tasks trigger until the current time is reached.

Leave your suite running for a while to confirm it is working as expected and then shut it down using the `stop` button in the `cylc` gui.

## Summary

- Clock triggers are a type of `dependency` which cause `tasks` to wait for the *wall-clock time* to reach the *cycle point* time.
- A clock trigger applies only to a single task.
- Clock triggers can only be used in datetime cycling suites.

For more information see the [Cylc User Guide](http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide) (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

### 3.4.2 Broadcast

This tutorial walks you through using `cylc broadcast` which can be used to change *task runtime configuration* (page 39) in a running suite, on-the-fly.

#### Purpose

`cylc broadcast` can be used to change any `[runtime]` setting whilst the suite is running.

The standard use of `cylc broadcast` is to update the suite to an unexpected change in configuration, for example modifying the host a task runs on.

#### Standalone Example

Create a new suite in the `cylc-run` directory called `tutorial-broadcast`:

```
mkdir ~/cylc-run/tutorial-broadcast
cd ~/cylc-run/tutorial-broadcast
```

Copy the following configuration into a `suite.rc` file:

```
[scheduling]
    initial cycle point = 1012
    [[dependencies]]
        [[[R1]]]
            graph = wipe_log => announce
        [[[PT1H]]]
            graph = announce[-PT1H] => announce

[runtime]
    [[wipe_log]]
        # Delete any files in the suite's "share" directory.
        script = rm "${CYLC_SUITE_SHARE_DIR}/knights" || true

    [[announce]]
        script = echo "${CYLC_TASK_CYCLE_POINT} - ${MESSAGE}" >> "${FILE}"
    [[environment]]
        WORD = ni
        MESSAGE = We are the knights who say \"${WORD}\"
        FILE = "${CYLC_SUITE_SHARE_DIR}/knights"
```

We now have a suite with an `announce` task which runs every hour, writing a message to a log file (`share/knights`) when it does so. For the first cycle the log entry will look like this:

```
10120101T0000Z - We are the knights who say "ni"!
```

The `cylc broadcast` command enables us to change runtime configuration whilst the suite is running. For instance we could change the value of the `WORD` environment variable using the command:

```
cylc broadcast tutorial-broadcast -n announce -s "[environment]WORD=it"
```

- `tutorial-broadcast` is the name of the suite.
- `-n announce` tells Cylc we want to change the runtime configuration of the `announce` task.
- `-s "[environment]WORD=it"` changes the value of the `WORD` environment variable to `it`.

Run the suite then try using the `cylc broadcast` command to change the message:

```
cylc run tutorial-broadcast
cylc broadcast tutorial-broadcast -n announce -s "[environment]WORD=it"
```

Inspect the share/knights file, you should see the message change at certain points.

Stop the suite:

```
cylc stop tutorial-broadcast
```

### In-Situ Example

We can call `cylc broadcast` from within a task's script. This effectively provides the ability for tasks to communicate between themselves.

It is almost always better for tasks to communicate using files but there are some niche situations where communicating via `cylc broadcast` is justified. This tutorial walks you through using `cylc broadcast` to communicate between tasks.

Add the following recurrence to the dependencies section:

```
[[[PT3H]]]
graph = announce[-PT1H] => change_word => announce
```

The `change_word` task runs the `cylc broadcast` command to randomly change the `WORD` environment variable used by the `announce` task.

Add the following runtime configuration to the runtime section:

```
[[[change_word]]
script = """
# Select random word.
IFS=',' read -r -a WORDS <<< $WORDS
WORD=${WORDS[$(date +%s) % ${#WORDS[@]}]}

# Broadcast random word to the announce task.
cylc broadcast $CYLC_SUITE_NAME -n announce -s "[environment]WORD=${WORD}"
"""

[[[environment]]]
WORDS = ni, it, ekke ekke ptang zoo boing
```

Run the suite and inspect the log. You should see the message change randomly after every third entry (because the `change_word` task runs every 3 hours) e.g:

```
10120101T0000Z - We are the knights who say "ni"!
10120101T0100Z - We are the knights who say "ni"!
10120101T0200Z - We are the knights who say "ni"!
10120101T0300Z - We are the knights who say "ekke ekke ptang zoo boing!"
```

Stop the suite:

```
cylc stop tutorial-broadcast
```

### 3.4.3 Family Triggers

To reduce duplication in the `graph` it is possible to write `dependencies` using collections of tasks called `families`).

This tutorial walks you through writing such dependencies using family `triggers`.

## Explanation

Dependencies between tasks can be written using a *qualifier* to describe the *task state* that the dependency refers to (e.g. succeed fail, etc). If a dependency does not use a qualifier then it is assumed that the dependency refers to the succeed state e.g:

```
bake_bread => sell_bread      # sell_bread is dependent on bake_bread
˓→succeeding.
bake_bread:succeed => sell_bread # sell_bread is dependent on bake_bread
˓→succeeding.
sell_bread:fail => through_away # through_away is dependent on sell_bread
˓→failing.
```

The left-hand side of a *dependency* (e.g. sell\_bread:fail) is referred to as the *trigger*.

When we write a trigger involving a family, special qualifiers are required to specify whether the dependency is concerned with *all* or *any* of the tasks in that family reaching the desired *state* e.g:

- succeed-all
- succeed-any
- fail-all

Such *triggers* are referred to as *family triggers*

Foo cylc gui bar

## Example

Create a new suite called tutorial-family-triggers:

```
mkdir ~/cylc-run/tutorial-family-triggers
cd ~/cylc-run/tutorial-family-triggers
```

Paste the following configuration into the suite.rc file:

```
[cylc]
    UTC mode = True # Ignore DST
[scheduling]
    [[dependencies]]
        graph = visit_mine => MINERS
[runtime]
    [[visit_mine]]
        script = sleep 5; echo 'off to work we go'

    [[MINERS]]
        script = """
sleep 5;
if ($RANDOM % 2)); then
    echo 'Diamonds!'; true;
else
    echo 'Nothing...'; false;
fi
"""
    [[doc, grumpy, sleepy, happy, bashful, sneezy, dopey]]
        inherit = MINERS
```

You have now created a suite that:

- Has a visit\_mine task that sleeps for 5 seconds then outputs a message.
- Contains a MINERS family with a command in it that randomly succeeds or fails.
- Has 7 tasks that inherit from the MINERS family.

Open the `cylc gui` then run the suite by pressing the “play” button (top left hand corner) then clicking *Start*:

```
cylc gui tutorial-family-triggers &
```

You should see the `visit_mine` task run, then trigger the members of the MINERS family. Note that some of the MINERS tasks may fail so you will need to stop your suite using the “stop” button in the `cylc gui` in order to allow it to shutdown.

## Family Triggering: Success

As you will have noticed by watching the suite run, some of the tasks in the MINERS family succeed and some fail.

We would like to add a task to sell any diamonds we find, but wait for all the miners to report back first so we only make the one trip.

We can address this by using *family triggers*. In particular, we are going to use the `finish-all` trigger to check for all members of the MINERS family finishing, and the `succeed-any` trigger to check for any of the tasks in the MINERS family succeeding.

Open your `suite.rc` file and change the `[[dependencies]]` to look like this:

```
[[dependencies]]
graph = """visit_mine => MINERS
          MINERS:finish-all & MINERS:succeed-any => sell_diamonds"""
```

Then, add the following task to the `[runtime]` section:

```
[[sell_diamonds]]
script = sleep 5
```

These changes add a `sell_diamonds` task to the suite which is run once all the MINERS tasks have finished and if any of them have succeeded.

Save your changes and run your suite. You should see the new `sell_diamonds` task being run once all the miners have finished and at least one of them has succeeded. As before, stop your suite using the “stop” button in the `cylc gui`.

## Family Triggering: Failure

Cylc also allows us to trigger off failure of tasks in a particular family.

We would like to add another task to close down unproductive mineshafts once all the miners have reported back and had time to discuss their findings.

To do this we will make use of family triggers in a similar manner to before.

Open your `suite.rc` file and change the `[[dependencies]]` to look like this:

```
[[dependencies]]
graph = """visit_mine => MINERS
          MINERS:finish-all & MINERS:succeed-any => sell_diamonds
          MINERS:finish-all & MINERS:fail-any => close_shafts
          close_shafts => !MINERS
          """
```

Alter the `[[sell_diamonds]]` section to look like this:

```
[[close_shafts, sell_diamonds]]
script = sleep 5
```

These changes add a `close_shafts` task which is run once all the MINERS tasks have finished and any of them have failed. On completion it applies a *suicide trigger* to the MINERS family in order to allow the suite to shutdown.

Save your changes and run your suite. You should see the new `close_shafts` run should any of the MINERS tasks be in the failed state once they have all finished.

---

**Tip:** See the *Suicide Triggers* (page 85) tutorial for handling task failures.

---

## Different Triggers

Other family *qualifiers* beyond those covered in the example are also available.

The following types of “all” qualifier are available:

- `:start-all` - all the tasks in the family have started
- `:succeed-all` - all the tasks in the family have succeeded
- `:fail-all` - all the tasks in the family have failed
- `:finish-all` - all the tasks in the family have finished

The following types of “any” qualifier are available:

- `:start-any` - at least one task in the family has started
- `:succeed-any` - at least one task in the family has succeeded
- `:fail-any` - at least one task in the family has failed
- `:finish-any` - at least one task in the family has finished

## Summary

- Family triggers allow you to write dependencies for collections of tasks.
- Like *task triggers*, family triggers can be based on success, failure, starting and finishing of tasks in a family.
- Family triggers can trigger off either *all* or *any* of the tasks in a family.

### 3.4.4 Inheritance

We have seen in the *runtime tutorial* (page 53) how tasks can be grouped into families.

In this tutorial we will look at nested families, inheritance order and multiple inheritance.

#### Inheritance Hierarchy

Create a new suite by running the command:

```
rose tutorial inheritance-tutorial  
cd ~/cylc-run/inheritance-tutorial
```

You will now have a `suite.rc` file that defines two tasks each representing a different aircraft, the Airbus A380 jumbo jet and the Robson R44 helicopter:



```

[scheduling]
  [[dependencies]]
    graph = a380 & r44

[runtime]
  [[VEHICLE]]
    init-script = echo 'Boarding'
    pre-script = echo 'Departing'
    post-script = echo 'Arriving'

  [[AIR_VEHICLE]]
    inherit = VEHICLE
    [[[meta]]]
      description = A vehicle which can fly.

  [[AIRPLANE]]
    inherit = AIR_VEHICLE
    [[[meta]]]
      description = An air vehicle with fixed wings.
    [[[environment]]]
      CAN TAKE OFF VERTICALLY = false

  [[HELICOPTER]]
    inherit = AIR_VEHICLE
    [[[meta]]]
      description = An air vehicle with rotors.
    [[[environment]]]
      CAN TAKE OFF VERTICALLY = true

  [[a380]]
    inherit = AIRPLANE
    [[[meta]]]
      title = Airbus A380 Jumbo-Jet.

  [[r44]]
    inherit = HELICOPTER
    [[[meta]]]

```

(continues on next page)

(continued from previous page)

```
title = Robson R44 Helicopter.
```

**Note:** The [meta] section is a freeform section where we can define metadata to be associated with a task, family or the suite itself.

This metadata should not be mistaken with Rose *Configuration Metadata* (page 206).

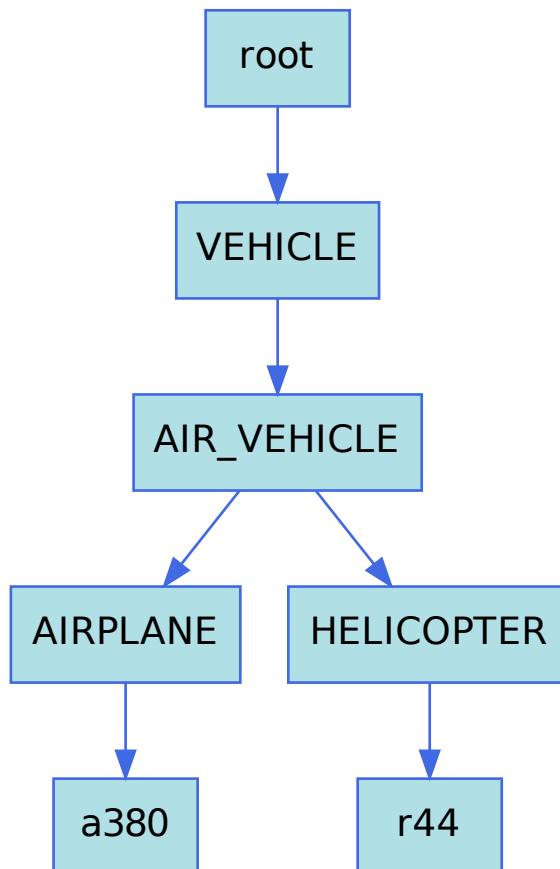
### Reminder

By convention we write family names in upper case (with the exception of the special root family) and task names in lower case.

These two tasks sit at the bottom of an inheritance tree. The cylc graph command has an option (-n) for drawing such inheritance hierarchies:

```
cylc graph -n . &
```

Running this command will generate the following output:



---

**Note:** The `root` family sits at the top of the inheritance tree as all tasks/families automatically inherit it:

---

Cyclc handles inheritance by starting with the root family and working down the inheritance tree applying each section in turn.

To see the resulting configuration for the `a380` task use the `cyclc get-config` command:

```
cyclc get-config . --sparse -i "[runtime][a380]"
```

You should see some settings which have been inherited from the VEHICLE and AIRPLANE families as well as a couple defined in the `a380` task.

```
init-script = echo 'Boarding'                                # Inherited from VEHICLE
pre-script = echo 'Departing'                               # Inherited from VEHICLE
post-script = echo 'Arriving'                               # Inherited from VEHICLE
inherit = AIRPLANE                                         # Defined in a380
[[[meta]]]
  description = An air vehicle with fixed wings.          # Inherited from AIR_VEHICLE -_
  → overwritten by AIRPLANE
    title = Airbus A380 Jumbo-Jet.                         # Defined in a380
[[[environment]]]
  CAN_TAKE_OFF_VERTICALLY = false                         # Inherited from AIRPLANE
```

Note that the `description` setting is defined in the `AIR_VEHICLE` family but is overwritten by the value specified in the `AIRPLANE` family.

## Multiple Inheritance

Next we want to add a vehicle called the V-22 Osprey to the suite. The V-22 is a cross between a plane and a helicopter - it has wings but can take-off and land vertically.



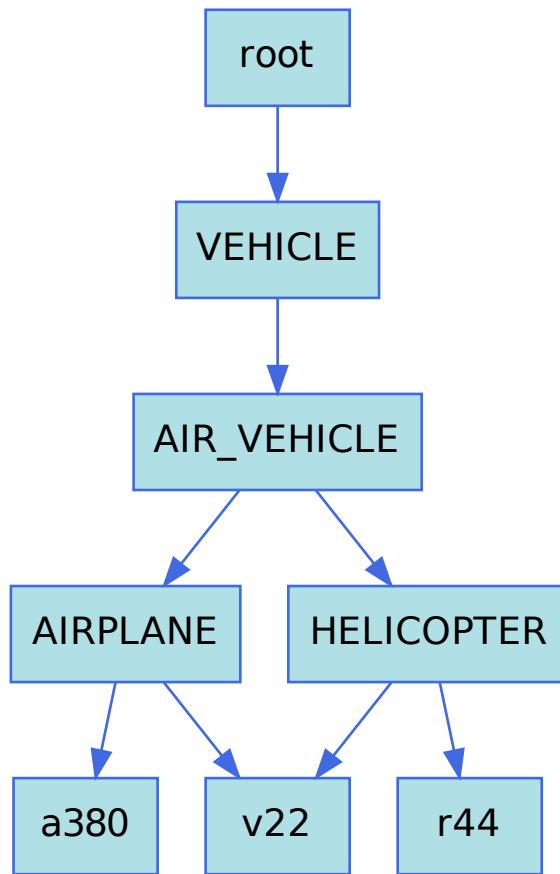
As the V-22 can be thought of as both a plane and a helicopter we want it to inherit from both the `AIRPLANE` and `HELICOPTER` families. In Cyclc we can inherit from multiple families by separating their names with commas:

Add the following task to your `suite.rc` file.

```
[[v22]]
  inherit = AIRPLANE, HELICOPTER
  [[[meta]]]
    title = V-22 Osprey Military Aircraft.
```

Refresh your `cyclc graph` window or re-run the `cyclc graph` command.

The inheritance hierarchy should now look like this:



Inspect the configuration of the `v22` task using the `cylc get-config` command.

#### Hint

```
cylc get-config . --sparse -i "[runtime][v22]"
```

You should see that the `CAN_TASK_OFF_VERTICALLY` environment variable has been set to `false` which isn't right. This is because of the order in which inheritance is applied.

Cylc handles multiple-inheritance by applying each family from right to left. For the `v22` task we specified `inherit = AIRPLANE, HELICOPTER` so the `HELICOPTER` family will be applied first and the `AIRPLANE` family after.

The inheritance order would be as follows:

```

root
VEHICLE
AIR_VEHICLE
HELICOPTER      # sets "CAN_TASK_OFF_VERTICALLY" to "true"
AIRPLANE        # sets "CAN_TASK_OFF_VERTICALLY" to "false"
v22
  
```

We could fix this problem by changing the order of inheritance:

```
inherit = HELICOPTER, AIRPLANE
```

Now the HELICOPTER family is applied second so its values will override any in the AIRPLANE family.

```
root
VEHICLE
AIR_VEHICLE
AIRPLANE      # sets "CAN_TAKE_OFF_VERTICALLY" to "false"
HELICOPTER    # sets "CAN_TAKE_OFF_VERTICALLY" to "true"
v22
```

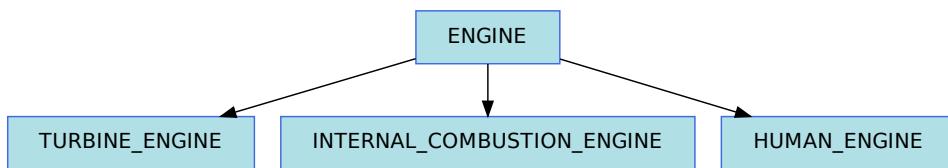
Inspect the configuration of the v22 task using `cylc get-config` to confirm this.

## More Inheritance

We will now add some more families and tasks to the suite.

### Engine Type

Next we will define four families to represent three different types of engine.



Each engine type should set an environment variable called FUEL which we will assign to the following values:

- Turbine - kerosene
- Internal Combustion - petrol
- Human - pizza

Add lines to the `runtime` section to represent these four families.

---

### Solution

```
[[[ENGINE]]
[[[TURBINE_ENGINE]]
  inherit = ENGINE
  [[[environment]]]
  FUEL = kerosene
[[[INTERNAL_COMBUSTION_ENGINE]]
  inherit = ENGINE
  [[[environment]]]
  FUEL = petrol
[[[HUMAN_ENGINE]]
  inherit = ENGINE
  [[[environment]]]
  FUEL = pizza
```

We now need to make the three aircraft inherit from one of the three engines. The aircraft use the following types of engine:

- A380 - turbine
- R44 - internal combustion
- V22 - turbine

Modify the three tasks so that they inherit from the relevant engine families.

---

### Solution

```
[[a380]]
    inherit = AIRPLANE, TURBINE_ENGINE
    [[[meta]]]
        title = Airbus A380 Jumbo-Jet.

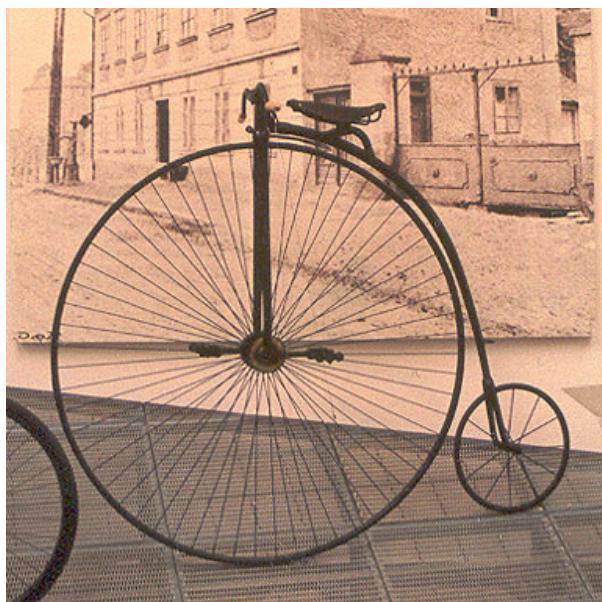
[[r44]]
    inherit = HELICOPTER, INTERNAL_COMBUSTION_ENGINE
    [[[meta]]]
        title = Robson R44 Helicopter.

[[v22]]
    inherit = AIRPLANE, HELICOPTER, TURBINE_ENGINE
    [[[meta]]]
        title = V-22 Osprey Military Aircraft.
```

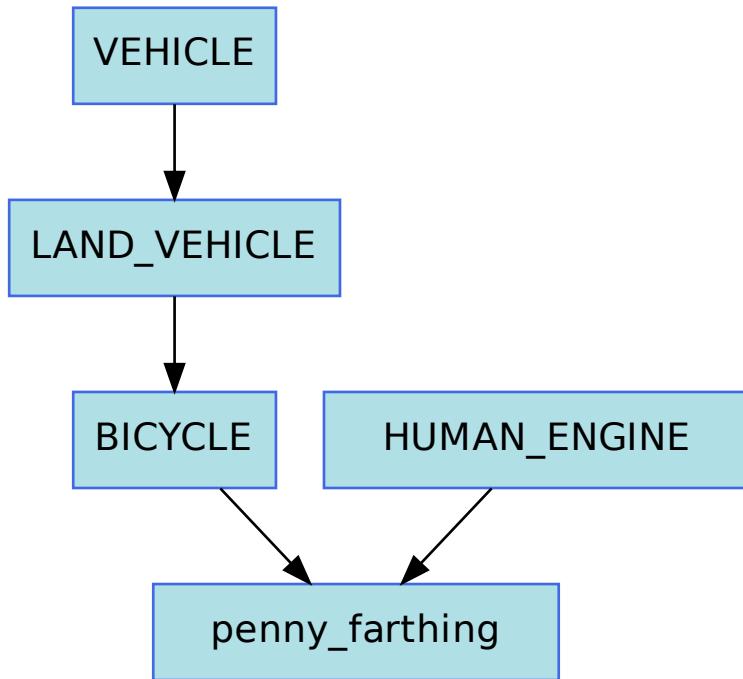
---

### Penny Farthing

Next we want to add a new type of vehicle, an old-fashioned bicycle called a penny farthing.



To do this we will need to add two new families, LAND\_VEHICLE and BICYCLE as well as a new task, penny\_farthing related in the following manner:



Add lines to the `runtime` section to represent the two new families and one task outlined above.

Add a description ([meta]description) to the `LAND_VEHICLE` and `BICYCLE` families and a title ([meta]title) to the `penny_farthing` task.

---

### Solution

```

[[LAND_VEHICLE]]
  inherit = VEHICLE
  [[[meta]]]
    description = A vehicle which can travel over the ground.

[[BICYCLE]]
  inherit = LAND_VEHICLE
  [[[meta]]]
    description = A small two-wheeled vehicle.

[[penny_farthing]]
  inherit = BICYCLE, HUMAN_ENGINE
  [[[meta]]]
    title = An old-fashioned bicycle.
  
```

Using `cylc get-config` to inspect the configuration of the `penny_farthing` task we can see that it inherits settings from the `VEHICLE`, `BICYCLE` and `HUMAN_ENGINE` families.

```

inherit = BICYCLE, HUMAN_ENGINE
init-script = echo 'Boarding' # Inherited from VEHICLE
pre-script = echo 'Departing' # Inherited from VEHICLE
post-script = echo 'Arriving' # Inherited from VEHICLE
[[[environment]]]
  
```

(continues on next page)

(continued from previous page)

```

FUEL = pizza           # Inherited from HUMAN_ENGINE
[[[meta]]]
description = A small two-wheeled vehicle. # Inherited from LAND_VEHICLE -_
˓→ overwritten by BICYCLE
title = An old-fashioned bicycle.          # Defined in penny_farthng

```

**Hint**

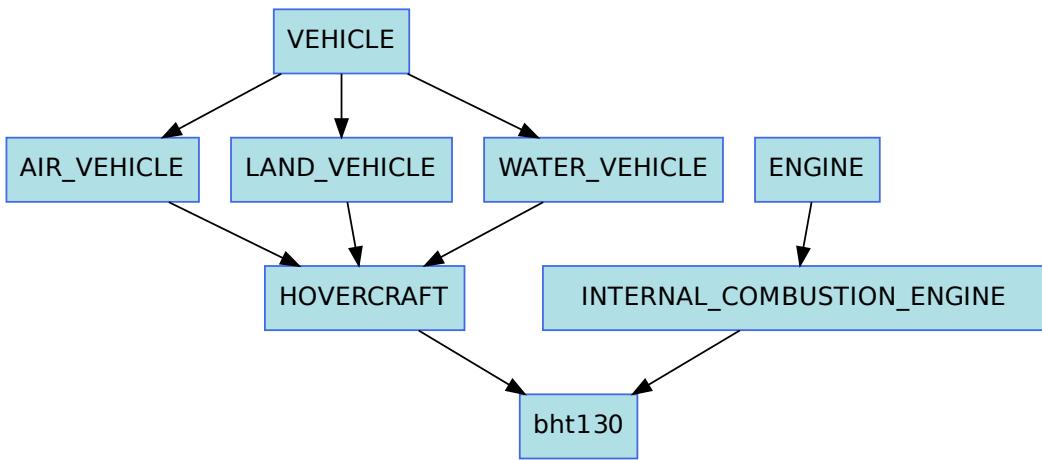
```
cyclc get-config . --sparse -i "[runtime]penny_farthng"
```

**Hovercraft**

We will now add a hovercraft called the Hoverwork BHT130, better known to some as the Isle Of Wight Ferry.



Hovercraft can move over both land and water and in some respects can be thought of as flying vehicles.



Write new families and one new task to represent the above structure.

Add a description ([meta]description) to the WATER\_VEHICLE and HOVERCRAFT families and a title ([meta]title) to the bht130 task.

**Solution**

```

[[WATER_VEHICLE]]
  inherit = VEHICLE
  [[[meta]]]
    description = A vehicle which can travel over water.

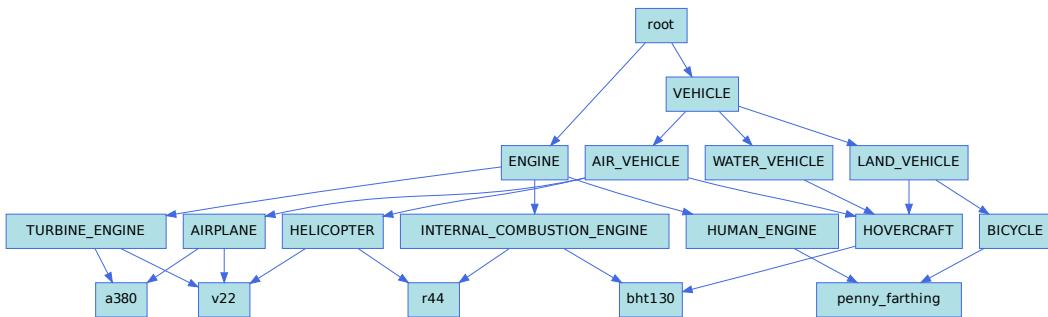
[[HOVERCRAFT]]
  inherit = LAND_VEHICLE, AIR_VEHICLE, WATER_VEHICLE
  [[[meta]]]
    description = A vehicle which can travel over ground, water and ice.

[[bht130]]
  inherit = HOVERCRAFT, INTERNAL_COMBUSTION_ENGINE
  [[[meta]]]
    title = Griffon Hoverwork BHT130 (Isle Of Whight Ferry).

```

## Finished Suite

You should now have a suite with an inheritance hierarchy which looks like this:



### 3.4.5 Queues

Queues are used to put a limit on the number of tasks that will be active at any one time, even if their dependencies are satisfied. This avoids swamping systems with too many tasks at once.

#### Example

In this example, our suite manages a particularly understaffed restaurant.

Create a new suite called `queues-tutorial`:

```

rose tutorial queues-tutorial
cd ~/cylc-run/queues-tutorial

```

You will now have a `suite.rc` file that looks like this:

```

[scheduling]
  [[dependencies]]
    graph =
      open_restaurant => steak1 & steak2 & pasta1 & pasta2 & pasta3 & \
                           pizza1 & pizza2 & pizza3 & pizza4

```

(continues on next page)

(continued from previous page)

```

steak1 => ice_cream1
steak2 => cheesecake1
pasta1 => ice_cream2
pasta2 => sticky_toffee1
pasta3 => cheesecake2
pizza1 => ice_cream3
pizza2 => ice_cream4
pizza3 => sticky_toffee2
pizza4 => ice_cream5
"""

[runtime]
  [[open_restaurant]]
  [[MAINS]]
  [[DESSERT]]
  [[steak1,steak2,pasta1,pasta2,pasta3,pizza1,pizza2,pizza3,pizza4]]
    inherit = MAINS
  [[ice_cream1,ice_cream2,ice_cream3,ice_cream4,ice_cream5]]
    inherit = DESSERT
  [[cheesecake1,cheesecake2,sticky_toffee1,sticky_toffee2]]
    inherit = DESSERT

```

**Note:** In graph sections backslash (\) is a line continuation character i.e. the following two examples are equivalent:

```
foo => bar & \
      baz
```

```
foo => bar & baz
```

Open the cylc gui then run the suite:

```
cylc gui queues-tutorial &
cylc run queues-tutorial
```

You will see that all the steak, pasta, and pizza tasks are run at once, swiftly followed by all the ice\_cream, cheesecake, sticky\_toffee tasks as the customers order from the dessert menu.

This will overwhelm our restaurant staff! The chef responsible for MAINS can only handle 3 tasks at any given time, and the DESSERT chef can only handle 2.

We need to add some queues. Add a [queues] section to the [scheduling] section like so:

```
[scheduling]
  [[queues]]
    [[[mains_chef_queue]]]
      limit = 3 # Only 3 mains dishes at one time.
      members = MAINS
    [[[dessert_chef_queue]]]
      limit = 2 # Only 2 dessert dishes at one time.
      members = DESSERT
```

Re-open the cylc gui if you have closed it and re-run the suite.

You should see that there are now never more than 3 active MAINS tasks running and never more than 2 active DESSERT tasks running.

The customers will obviously have to wait!

## Further Reading

For more information, see the [Cylc User Guide](http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide) (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

### 3.4.6 Retries

Retries allow us to automatically re-submit tasks which have failed due to failure in submission or execution.

#### Purpose

Retries can be useful for tasks that may occasionally fail due to external events, and are routinely fixable when they do - an example would be a task that is dependent on a system that experiences temporary outages.

If a task fails, the Cylc retry mechanism can resubmit it after a pre-determined delay. An environment variable, `$CYLC_TASK_TRY_NUMBER` is incremented and passed into the task - this means you can write your task script so that it changes behaviour accordingly.

#### Example



Create a new suite by running the following commands:

```
rose tutorial retries-tutorial
cd retries-tutorial
```

You will now have a suite with a `roll_doubles` task which simulates trying to roll doubles using two dice:

```
[cylc]
    UTC mode = True # Ignore DST

[scheduling]
    [[dependencies]]
        graph = start => roll_doubles => win

[runtime]
    [[start]]
    [[win]]
    [[roll_doubles]]
        script = """
sleep 10
RANDOM=$$ # Seed $RANDOM
DIE_1=$((RANDOM%6 + 1))
DIE_2=$((RANDOM%6 + 1))
echo "Rolled $DIE_1 and $DIE_2..."
if (($DIE_1 == $DIE_2)); then
    echo "doubles!"
else
```

(continues on next page)

(continued from previous page)

```
    exit 1
fi
"""
```

## Running Without Retries

Let's see what happens when we run the suite as it is. Open the `cylc gui`:

```
cylc gui retries-tutorial &
```

Then run the suite:

```
cylc run retries-tutorial
```

Unless you're lucky, the suite should fail at the `roll_doubles` task.

Stop the suite:

```
cylc stop retries-tutorial
```

## Configuring Retries

We need to tell Cylc to retry it a few times - replace the line `[[roll_doubles]]` in the `suite.rc` file with:

```
[[roll_doubles]]
  [[[job]]]
    execution retry delays = 5*PT6S
```

This means that if the `roll_doubles` task fails, Cylc expects to retry running it 5 times before finally failing. Each retry will have a delay of 6 seconds.

We can apply multiple retry periods with the `execution retry delays` setting by separating them with commas, for example the following line would tell Cylc to retry a task four times, once after 15 seconds, then once after 10 minutes, then once after one hour then once after three hours.

```
execution retry delays = PT15S, PT10M, PT1H, PT3H
```

## Running With Retries

If you closed it, re-open the `cylc gui`:

```
cylc gui retries-tutorial &
```

Re-run the suite:

```
cylc run retries-tutorial
```

What you should see is Cylc retrying the `roll_doubles` task. Hopefully, it will succeed (there is only about a 1 in 3 chance of every task failing) and the suite will continue.

## Altering Behaviour

We can alter the behaviour of the task based on the number of retries, using `$CYLC_TASK_TRY_NUMBER`.

Change the `script` setting for the `roll_doubles` task to this:

```

sleep 10
RANDOM=$$ # Seed $RANDOM
DIE_1=$((RANDOM%6 + 1))
DIE_2=$((RANDOM%6 + 1))
echo "Rolled $DIE_1 and $DIE_2..."
if (($DIE_1 == $DIE_2)); then
    echo "doubles!"
elif ((CYLC_TASK_TRY_NUMBER >= 2)); then
    echo "look over there! ..."
    echo "doubles!" # Cheat!
else
    exit 1
fi

```

If your suite is still running, stop it, then run it again.

This time, the task should definitely succeed before the third retry.

## Further Reading

For more information see the [Cylc User Guide](http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide) (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

### 3.4.7 Suicide Triggers

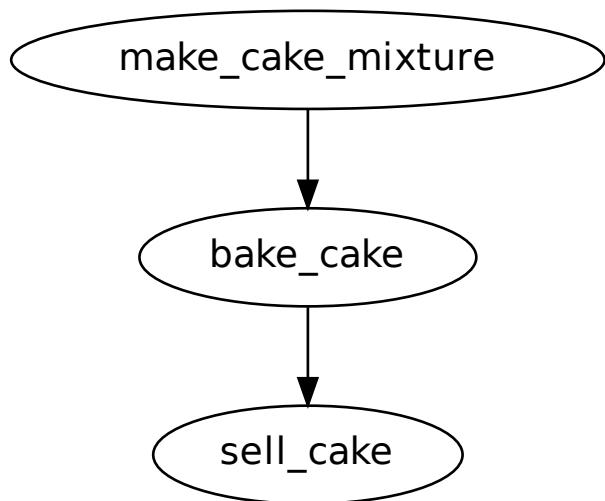
Suicide triggers allow us to remove a task from the suite's graph whilst the suite is running.

The main use of suicide triggers is for handling failures in the workflow.

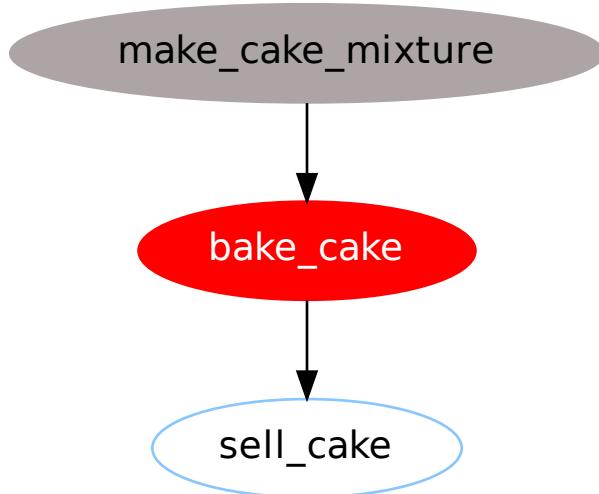
## Stalled Suites

Imagine a bakery which has a workflow that involves making cake.

```
make_cake_mixture => bake_cake => sell_cake
```



There is a 50% chance that the cake will turn out fine, and a 50% chance that it will get burnt. In the case that we burn the cake the workflow gets stuck.



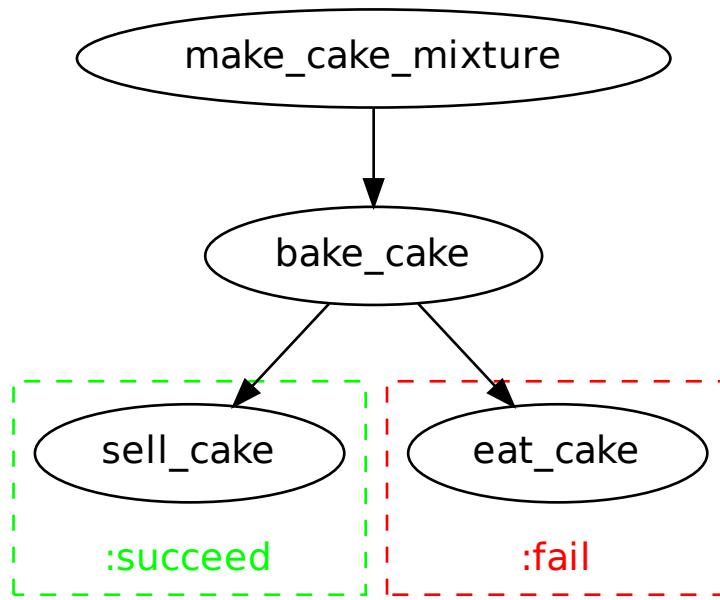
In this event the `sell_cake` task will be unable to run as it depends on `bake_cake`. We would say that this suite has *stalled*. When Cycl detects that a suite has stalled it sends you an email to let you know that the suite has got stuck and requires human intervention to proceed.

### Handling Failures

In order to prevent the suite from entering a stalled state we need to handle the failure of the `bake_cake` task.

At the bakery if they burn a cake they eat it and make another.

The following diagram outlines this workflow with its two possible pathways, `:succeed` in the event that `bake_cake` is successful and `:fail` otherwise.



We can add this logic to our workflow using the `fail` *qualifier*.

```

bake_cake => sell_cake
bake_cake:fail => eat_cake

```

### Reminder

If you don't specify a qualifier CycL assumes you mean `:succeed` so the following two lines are equivalent:

```

foo => bar
foo:succeed => bar

```

## Why Do We Need To Remove Tasks From The Graph?

Create a new suite called `suicide-triggers`:

```

mkdir -p ~/cylc-run/suicide-triggers
cd ~/cylc-run/suicide-triggers

```

Paste the following code into the `suite.rc` file:

```

[scheduling]
cycling mode = integer
initial cycle point = 1
[[dependencies]]
[[[P1]]]
graph =
    make_cake_mixture => bake_cake => sell_cake
    bake_cake:fail => eat_cake
"""

```

(continues on next page)

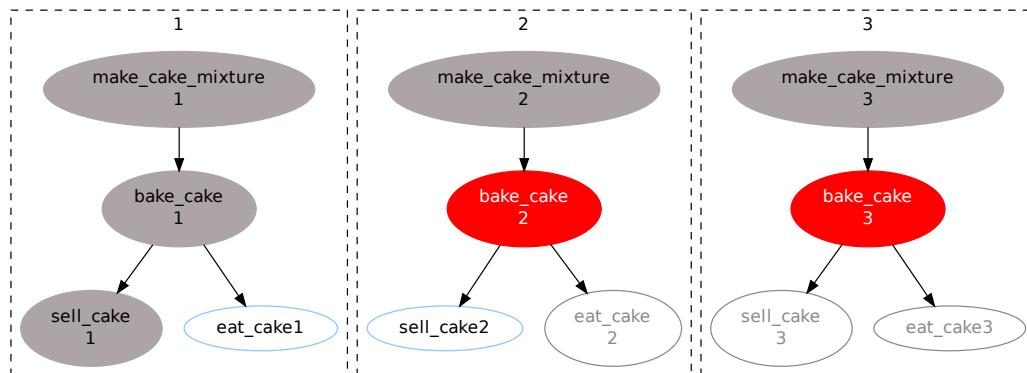
(continued from previous page)

```
[runtime]
  [[root]]
    script = sleep 2
  [[bake_cake]]
    # Random outcome 50% chance of success 50% chance of failure.
    script = sleep 2; if (( $RANDOM % 2 )); then true; else false; fi
```

Open the cylc gui and run the suite:

```
cylc gui suicide-triggers &
cylc run suicide-triggers
```

The suite will run for three cycles then get stuck. You should see something similar to the diagram below. As the bake\_cake task fails randomly what you see might differ slightly. You may receive a “suite stalled” email.



The reason the suite stalls is that, by default, Cylc will run a maximum of three cycles concurrently. As each cycle has at least one task which hasn't either succeeded or failed Cylc cannot move onto the next cycle.

---

**Tip:** For more information search `max active cycle points` in the [Cylc User Guide](#) (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>).

---

You will also notice that some of the tasks (e.g. `eat_cake` in cycle 2 in the above example) are drawn in a faded gray. This is because these tasks have not yet been run in earlier cycles and as such cannot run.

## Removing Tasks From The Graph

In order to get around these problems and prevent the suite from stalling we must remove the tasks that are no longer needed. We do this using suicide triggers.

A suicide trigger is written like a normal dependency but with an exclamation mark in-front of the task on the right-hand-side of the dependency meaning “*remove the following task from the graph at the current cycle point.*”

For example the following `graph string` would remove the task `bar` from the graph if the task `foo` were to succeed.

```
foo => ! bar
```

There are three cases where we would need to remove a task in the cake-making example:

1. If the `bake_cake` task succeeds we don't need the `eat_cake` task so should remove it.

```
bake_cake => ! eat_cake
```

2. If the `bake_cake` task fails we don't need the `sell_cake` task so should remove it.

```
bake_cake:fail => ! sell_cake
```

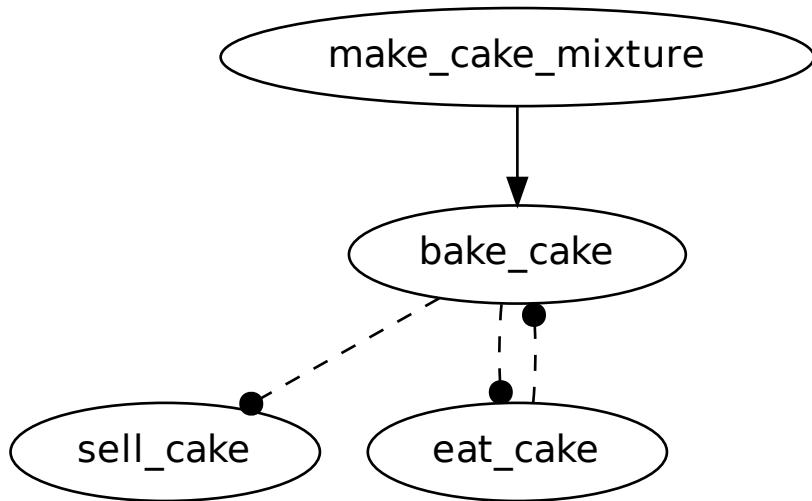
3. If the `bake_cake` task fails then we will need to remove it else the suite will stall. We can do this after the `eat_cake` task has succeeded.

```
eat_cake => ! bake_cake
```

Add the following three lines to the suite's graph:

```
bake_cake => ! eat_cake
bake_cake:fail => ! sell_cake
eat_cake => ! bake_cake
```

We can view suicide triggers in `cylc` graph by un-selecting the *Ignore Suicide Triggers* button in the toolbar. Suicide triggers will then appear as dashed lines with circular endings. You should see something like this:



## Downstream Dependencies

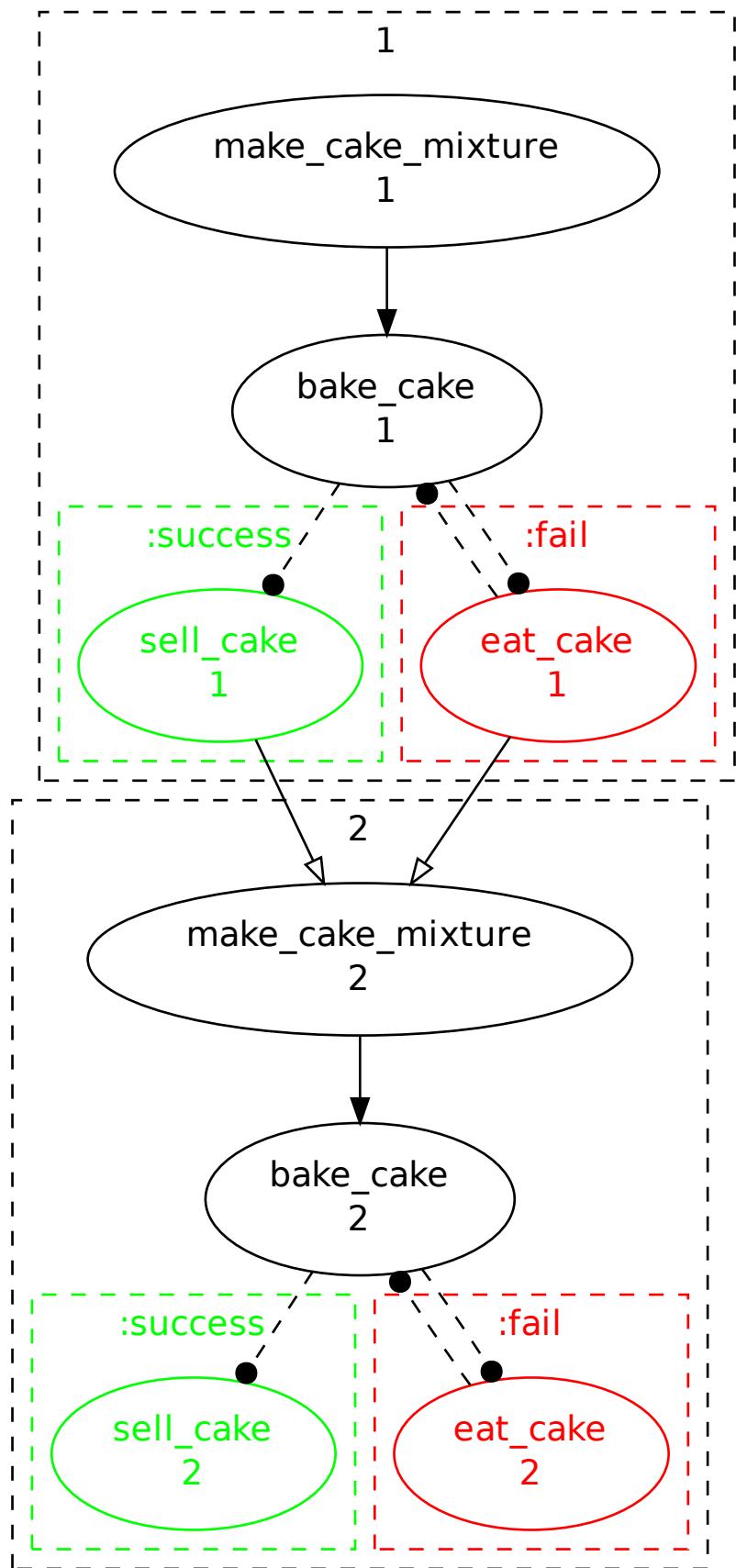
If we wanted to make the cycles run in order we might write an *inter-cycle dependency* like this:

```
sell_cake[-P1] => make_cake_mixture
```

In order to handle the event that the `sell_cake` task has been removed from the graph by a suicide trigger we can write our dependency with an or symbol `|` like so:

```
eat_cake[-P1] | sell_cake[-P1] => make_cake_mixture
```

Now the `make_cake_mixture` task from the next cycle will run after whichever of the `sell_cake` or `eat_cake` tasks is run.



Add the following *graph string* to your suite.

```
eat_cake[-P1] | sell_cake[-P1] => make_cake_mixture
```

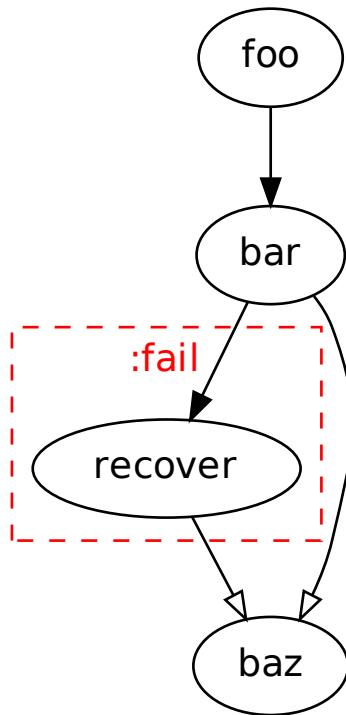
Open the `cyclc gui` and run the suite. You should see that if the `bake_cake` task fails both it and the `sell_cake` task disappear and are replaced by the `eat_cake` task.

## Variations

The following sections outline examples of how to use suicide triggers.

### Recovery Task

A common use case where a `recover` task is used to handle a task failure.



```
[scheduling]
[[dependencies]]
graph = """
# Regular graph.
foo => bar

# The fail case.
bar:fail => recover

# Remove the "recover" task in the success case.
bar => ! recover

# Remove the "bar" task in the fail case.

```

(continues on next page)

(continued from previous page)

```

recover => ! bar

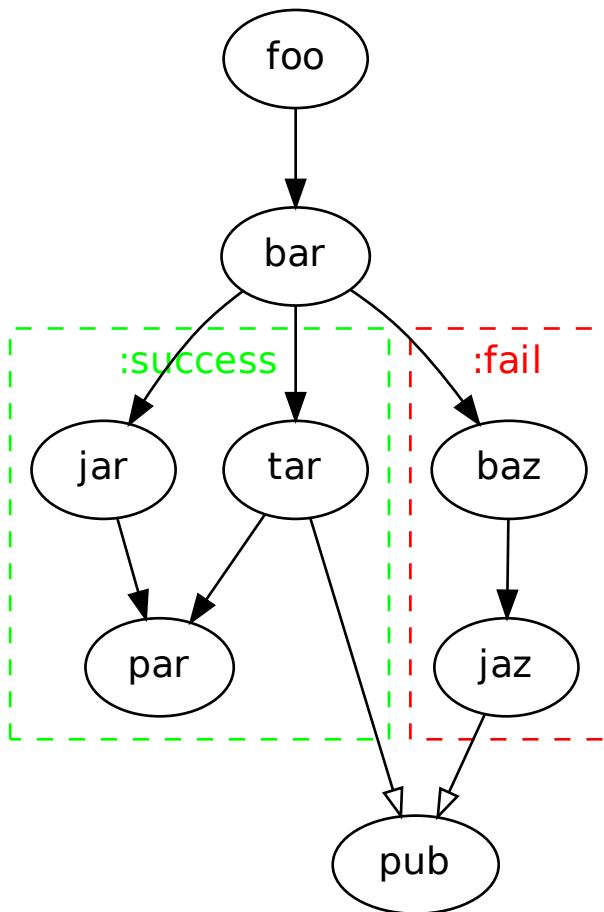
# Downstream dependencies.
bar | recover => baz
"""

[runtime]
[[root]]
script = sleep 1
[[bar]]
script = false

```

## Branched Workflow

A workflow where sub-graphs of tasks are to be run in the success and or fail cases.



```

[scheduling]
[[dependencies]]
graph = """
# Regular graph.
foo => bar

```

(continues on next page)

(continued from previous page)

```

# Success case.
bar => tar & jar

# Fail case.
bar:fail => baz => jaz

# Remove tasks from the fail branch in the success case.
bar => ! baz & ! jaz

# Remove tasks from the success branch in the fail case.
bar:fail => ! tar & ! jar & ! par

# Remove the bar task in the fail case.
baz => ! bar

# Downstream dependencies.
tar | jaz => pub
"""

[runtime]
[[root]]
script = sleep 1
[[bar]]
script = true

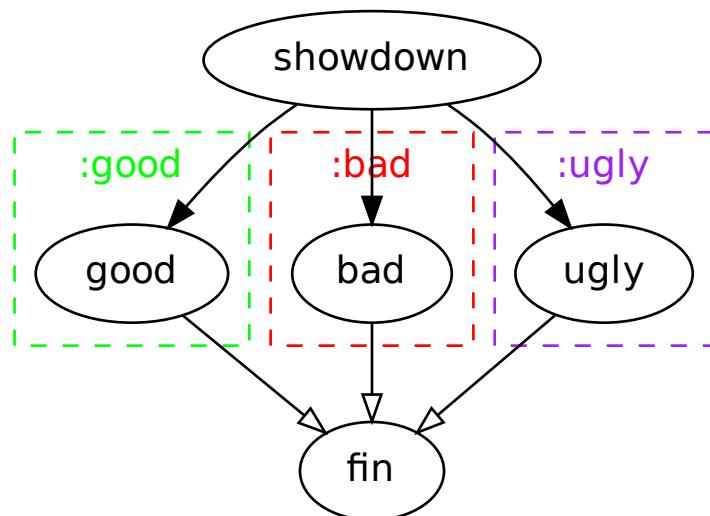
```

## Triggering Based On Other States

In these examples we have been using suicide triggers to handle task failure. The suicide trigger mechanism works with other qualifiers as well for example:

```
foo:start => ! bar
```

Suicide triggers can also be used with custom outputs. In the following example the task `showdown` produces one of three possible custom outputs, `good`, `bad` or `ugly`.



```
[scheduling]
  [[dependencies]]
    [[[R1]]]
      graph = """
        # In the "good" case run "good" but not "bad" or "ugly".
        showdown:good => good & ! bad & ! ugly

        # In the "bad" case run "bad" but not "good" or "ugly".
        showdown:bad => bad & ! good & ! ugly

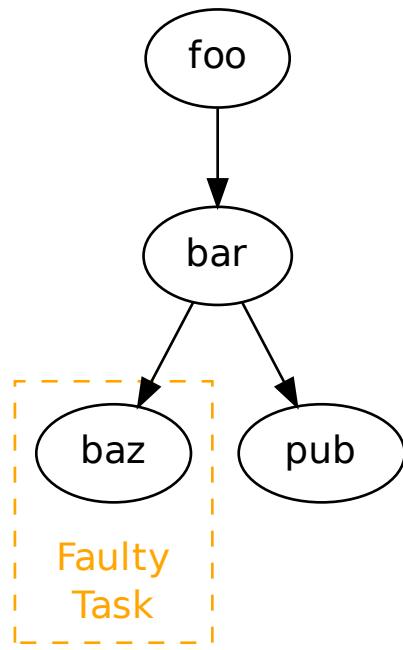
        # In the "ugly" case run "ugly" but not "good" or "bad".
        showdown:ugly => ugly & ! good & ! bad

        good | bad | ugly => fin
      """

[runtime]
  [[root]]
    script = sleep 1
  [[showdown]]
    # Randomly return one of the three custom outputs.
    script = """
      SEED=$RANDOM
      if ! (( $SEED % 3 )); then
        cylc message 'The-Good'
      elif ! (( ( $SEED + 1 ) % 3 )); then
        cylc message 'The-Bad'
      else
        cylc message 'The-Ugly'
      fi
    """
  [[[outputs]]]
    # Register the three custom outputs with cylc.
    good = 'The-Good'
    bad = 'The-Bad'
    ugly = 'The-Ugly'
```

## Self-Suiciding Task

An example of a workflow where there are no tasks which are dependent on the task to suicide trigger.



It is possible for a task to suicide trigger itself e.g:

```
foo:fail => ! foo
```

**Warning:** This is usually not recommended but in the case where there are no tasks dependent on the one to remove it is an acceptable approach.

```

[scheduling]
  [[dependencies]]
    graph =
      foo => bar => baz
      bar => pub

      # Remove the "pub" task in the event of failure.
      pub:fail => ! pub
    """
  """

[runtime]
  [[root]]
    script = sleep 1
  [[pub]]
    script = false
  
```



## ROSE TUTORIAL

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.

### 4.1 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 Ninja2 variables for use in the `suite.rc` file.

#### 4.1.1 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](https://en.wikipedia.org/wiki/INI_file)) file format. Like the file format for *Cylc suites* (page 14):

- 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](#) (page 14):

- 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.

### 4.1.2 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 [rose config-edit](#) (page 250) GUI which makes use of metadata for display and on-the-fly validation purposes.

## 4.2 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.

### 4.2.1 Application Configuration

An application configuration is a directory containing a *rose-app.conf* (page 194) 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* (page 194) setting e.g.:

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

Environment variables are specified inside the *rose-app.conf[env]* (page 194) 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.

### 4.2.2 Running Rose Applications

An application can be run using the *rose app-run* (page 245) 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` (page 194) file.

The `rose-app.conf` (page 194) file needs to define the command to run. Create a `rose-app.conf` (page 194) 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]` (page 194) 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* (page 34) 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]` (page 194) 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=
# 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` (page 194):

```
# 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.

`rose app-run` (page 245) 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.

## 4.3 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 *rose config-edit* (page 250) GUI.

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

### 4.3.1 The Metadata Format

Metadata is written in a *rose-meta.conf* (page 207) 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* (page 207) file uses the standard *Rose configuration format* (page 97).

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 3: 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 4: 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.

### 4.3.2 Metadata Commands

The *rose metadata-check* (page 256) 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 *rose macro* (page 254) 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']
```

### 4.3.3 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 \[SETTING\]`](#) (page 207).

#### 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 [`rose config-edit`](#) (page 250).

The [`rose config-edit`](#) (page 250) command opens a GUI which displays Rose configurations. Open the `metadata-tutorial` app:

```
rose config-edit &
```

**Tip:** Note [`rose config-edit`](#) (page 250) 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`](#) (page 194) 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`](#) (page 194) file. Hover the mouse over the hash to reveal the comment.

Keep the [`rose config-edit`](#) (page 250) 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`](#) (page 207) file:

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

In the [`rose-app.conf`](#) (page 194) file there are comments associated with each setting. Take these comments out of the [`rose-app.conf`](#) (page 194) 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 [rose metadata-check](#) (page 256):

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

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

Next reload the metadata in the [rose config-edit](#) (page 250) 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](#) (page 209) 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 [rose config-edit](#) (page 250) 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 [rose config-edit](#) (page 250) window. The INTERVAL setting should now appear as an integer rather than a text field.

In the [rose config-edit](#) (page 250) 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 *rose config-edit* (page 250) 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* (page 194) 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* (page 207) 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* (page 194) file) we will need to close and reload the *rose config-edit* (page 250) 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* (page 210) 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* (page 207) file:

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

Validate the metadata and refresh the *rose config-edit* (page 250) 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* (page 210) metadata option can be used to provide a longer description of each option.

---

## 7. Validate with rose macro.

On the command line *rose macro* (page 254) can be used to check that the configuration is compliant with the metadata. Try editing the *rose-app.conf* (page 194) file to introduce errors then validating the configuration by running:

```
rose macro -V
```

## 4.4 Rose Suite Configurations

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

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

#### 4.4.1 Configuration Format

A Rose suite configuration is a CycL *suite directory* containing a *rose-suite.conf* (page 196) file.

The *rose-suite.conf* (page 196) file is written in the same *format* (page 97) as the *rose-app.conf* (page 194) file. Its main configuration sections are:

***rose-suite.conf[env]* (page 196)** Environment variables for use by the whole suite.

***rose-suite.conf[jinja2:suite.rc]* (page 197)** *Jinja2* (<http://jinja.pocoo.org/>) variables for use in the *suite.rc* file.

***rose-suite.conf[file:NAME]* (page 197)** 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* (page 196) file. These variables can then be used in the *suite.rc* file:

Listing 5: *rose-suite.conf*

```
[env]
GREETING=Hello

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

Listing 6: *suite.rc*

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

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

#### 4.4.2 Suite Directory Vs Run Directory

*suite directory* The directory in which the suite is written. The *suite.rc* and *rose-suite.conf* (page 196) files live here.

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

Throughout the *Cyclc Tutorial* (page 13) we wrote suites in the *cyclc-run* directory. As Cyclc runs suites in the *cyclc-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 Cyclc it is possible to separate the *suite directory* and *run directory* using the *cyclc 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.

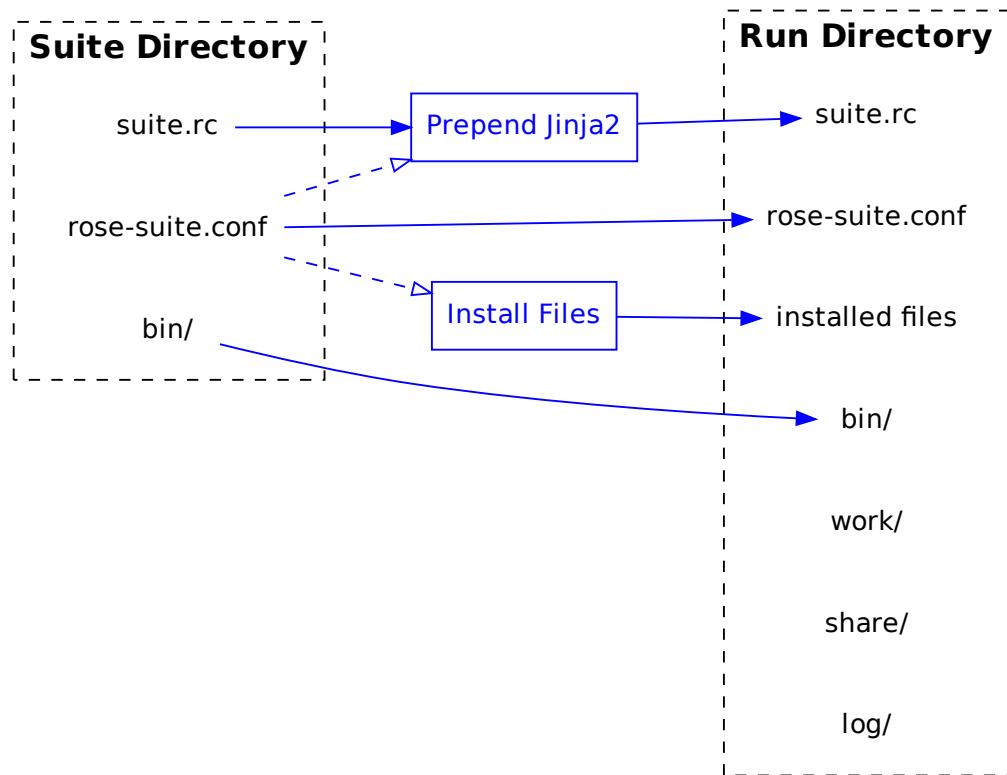
---

#### 4.4.3 Running Rose Suite Configurations

Rose *Application Configurations* (page 99) are run using *rose app-run* (page 245), Rose Suite Configurations are run using *rose suite-run* (page 263).

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`](#) (page 196) file are installed.
3. Jinja2 variables defined in the [`rose-suite.conf`](#) (page 196) file are added to the top of the `suite.rc` file.
4. The Cycl suite is validated.
5. The Cycl suite is run.
6. The Cycl GUI is launched.



Like [`rose app-run`](#) (page 245), [`rose suite-run`](#) (page 263) 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 107)).

#### 4.4.4 Start, Stop, Restart

Under Rose, suites will run using the name of the suite directory. For instance if you run [`rose suite-run`](#) (page 263) 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>
```

**Running/Interacting With Suites (page 175)** Suites must be run using the [rose suite-run](#) (page 263) command which in turn calls the `cylc run` command.

**Stopping Suites (page 175)** Suites can be stopped using the `cylc stop <SUITE_NAME>` command, as for regular Cycl suites.

**Restarting Suites (From Stopped) (page 175)** There are two options for restarting:

- To pick up where the suite left off use [rose suite-restart](#) (page 262). 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 [rose suite-run](#) (page 263) command.

See the [Cheat Sheet](#) (page 175) for more information.

---

**Note:** [rose suite-run](#) (page 263) 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 [rose suite-run](#) (page 263) 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](#) (page 49) 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](#) (page 49).

### 2. Create A Rose Suite Configuration.

Create a blank [rose-suite.conf](#) (page 196) file:

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

You now have a Rose suite configuration. A [rose-suite.conf](#) (page 196) file does not need to have anything in it but it is required to run [rose suite-run](#) (page 263).

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](#) (page 196) 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/) (<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` (page 196) 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 102).

---

### 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 `rose config-edit` (page 250) 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` (page 196) 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/) (<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.
```

(continues on next page)

(continued from previous page)

```

-     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 [rose suite-run](#) (page 263) 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.

### 4.4.5 Rose Applications In Rose Suite Configurations

In Cyc suites, Rose applications are placed in an app/ directory which is copied across to the [run directory](#) with the rest of the suite by [rose suite-run](#) (page 263) when the suite configuration is run.

When we run Rose applications from within Cyc suites we use the [rose task-run](#) (page 267) command rather than the [rose app-run](#) (page 245) command.

When run, [rose task-run](#) (page 267) searches for an application with the same name as the Cyc task in the app/ directory.

The [rose task-run](#) (page 267) command also interfaces with Cyc to provide a few useful environment variables (see the [command-line reference](#) (page 267) for details). The application will run in the [work directory](#), just like for a regular Cyc task.

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

Listing 7: suite.rc

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

Listing 8: 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](#) (page 278) environment variable. For example the greetings task will run the hello app in the task defined below.

Listing 9: suite.rc

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

## 4.4.6 Rose Bush

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



The screenshot shows a table with the following columns: task status, job status, cycle point, task name, job #, submit time, queue Δt, run Δt, job host, job batch, and job logs. The table contains five rows of data:

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

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

If a Rose Bush server is provided at your site, you can open the Rose Bush page for a suite by running the [rose suite-log](#) (page 262) command in the suite directory.

Otherwise an add-hoc web server can be set up using the [rose bush](#) (page 246) start command argument.

---

### 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 Cycl 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](#) (page 225) 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 134). 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](#) (page 194) 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 `rose app-run` (page 245) 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` (page 194) 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` (page 194) 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
```

(continues on next page)

(continued from previous page)

```
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 `rose task-run` (page 267). 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 `rose config-edit` (page 250) 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.

---

## 4.5 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/) ([http://metomi.github.io/fcm/doc/user\\_guide/](http://metomi.github.io/fcm/doc/user_guide/)).

### 4.5.1 Rosie Suites

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

Rosie suites can be created by the command:

**rosie create (page 269)** 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`.

### 4.5.2 Version Control

In Rosie suites the `suite directory` is added to `version control` (page 114) 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/](http://metomi.github.io/fcm/doc/user_guide/)) for more information.

### 4.5.3 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` (page 204) setting.

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

### 4.5.4 The `rose-suite.info` File

All Rosie suites require a `rose-suite.info` (page 197) file. This file provides information about the suite for use in the suite management and version control systems. The `rose-suite.info` (page 197) file uses the `Rose Configuration Format` (page 97). 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.

### 4.5.5 Managing Suites

Rosie provides commands for managing suites, including:

***rosie checkout*** (page 268) Creates a local copy of a suite.

***rosie ls*** (page 273) Lists all locally checked-out suites.

***rosie lookup*** (page 273) Searches the suite database (using information from suite's *rose-suite.info* (page 197) files).

Rosie also provides a GUI called *rosie go* (page 270) which incorporates the functionality of the above commands.

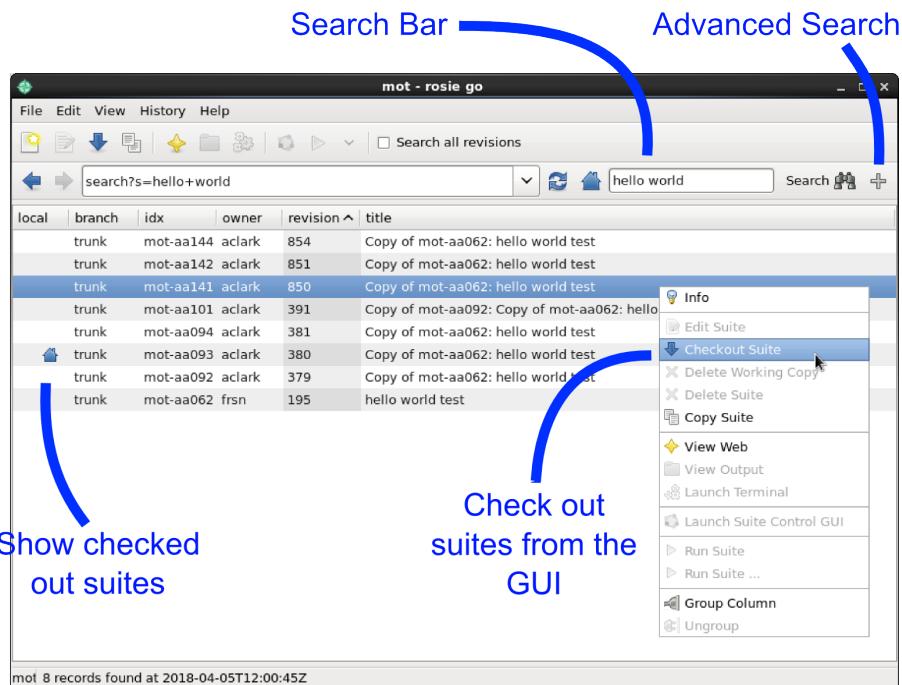


Fig. 2: Screenshot of the *rosie go* GUI.

---

#### 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.

---

##### 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* (page 197) 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 `rosie go` (page 270) 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` (page 197) 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 *rose config-edit* (page 250) GUI or a text editor, change the RESOLUTION setting in the *rose-suite.conf* (page 196) file to 0 . 1.

Check the status of the project:

```
fcm st
```

You should see the *rose-suite.conf* (page 196) 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* (page 196) file) then commit the merge:

```
fcm st  
fcm ci
```

---

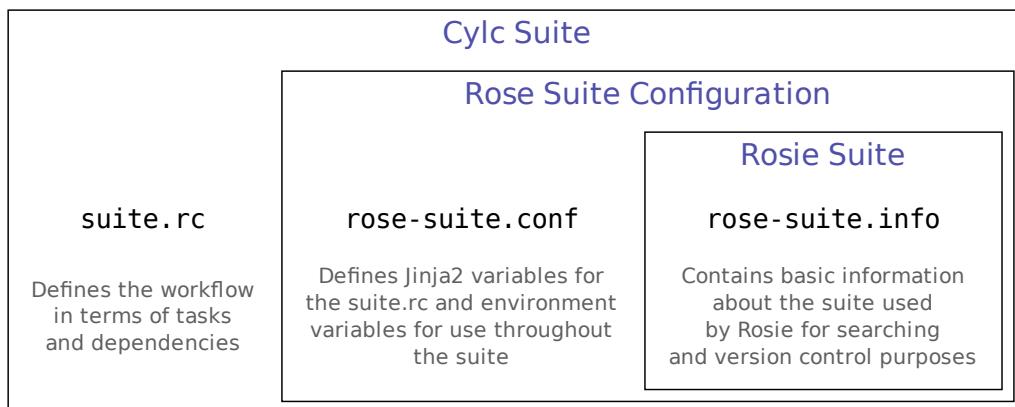
## 4.6 Summary

### 4.6.1 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` (page 194) file.

### 4.6.2 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:

`rose app-run` (page 245) Runs a Rose application.

`rose task-run` (page 267) Runs a Rose application from within a CycL suite.

`rose suite-run` (page 263) Runs a Rose suite.

`rose suite-restart` (page 262) Runs a Rose suite, picking up where it left off from the previous run.

The CycL commands do not know about the `rose-suite.conf` (page 196) file so for Rose suite configurations you will have to install the suite before using commands such as `cyclc graph`, e.g:

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

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

### 4.6.3 Rose Utilities

Rose contains some utilities to make life easier:

`rose date` (page 250) A utility for parsing, manipulating and formatting date-times which is useful for working with the CycL *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 121) for more information.

`rose host-select` (page 253) 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}` (page 201) 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 `rose host-select` (page 253) command can be used within CycL suites to determine which host a task runs on:

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

### 4.6.4 Rose Built-In Applications

Along with Rose utilities there are also *Rose built-in applications* (page 283).

**fcm\_make** (page 284) A template for running the fcm make command.

**rose\_ana** (page 286) Runs the rose-ana analysis engine.

**rose\_arch** (page 289) Provides a generic solution to configure site-specific archiving of suite files.

**rose\_bunch** (page 291) For the running of multiple command variants in parallel under a single job.

**rose\_prune** (page 293) A framework for housekeeping a cycling suite.

## 4.6.5 Next Steps

**Further Topics** (page 120) Tutorials going over some of the more specific aspects of Rose not covered in the main tutorial.

**Cheat Sheet** (page 175) A quick breakdown of the commands for running and interacting with suites using Cyc and Rose.

**Command Reference** (page 245) Contains the command-line documentation (also obtainable by calling `rose --help`).

**Rose Configuration** (page 193) The possible settings which can be used in the different Rose configuration files.

**Cyc Suite Design Guide** (<http://cylc.github.io/cylc/doc/suite-design-guide.pdf>) Contains recommended best practice for the style and structure of Cyc suites.

## 4.7 Further Topics

This section goes into detail in additional Rose topics.

### 4.7.1 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 `rose task-run` (page 267).

#### 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` (page 196) 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` (page 194) 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 `rose suite-run` (page 263).

Once it has finished use `rose suite-log` (page 262) 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”.

## Adding Alternative Commands

Open the `rose-app.conf` (page 194) 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.

## 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

### 4.7.2 Date and Time Manipulation

*Datetime cycling* suites inevitably involve performing some form of datetime arithmetic. In the *weather forecasting suite* (page 34) we wrote in the CycL tutorial this arithmetic was done using the `cyclc cyclepoint` command. For example we calculated the cycle point three hours before the present cycle using:

```
cyclc cyclepoint --offset-hours=-3
```

Rose provides the [rose date](#) (page 250) command which provides functionality beyond `cyclc cyclepoint` as well as the [ROSE\\_DATAC](#) (page 276) environment variable which provides an easy way to get the path of the share/cycle directory.

### The `rose date` Command

The [rose date](#) (page 250) 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 [rose date](#) (page 250) command reference for more information.

### Using `rose date` In A Suite

In datetime cycling suites [rose date](#) (page 250) 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 [rose task-env](#) (page 265) then [rose date](#) (page 250) 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'
```

### 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 [rose task-env](#) (page 265) command provides the environment variable `ROSE_DATAC` (page 276) 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 [rose task-env](#) (page 265) e.g:

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

The path is then made available as the `ROSE_DATACT1H` environment variable.

### 4.7.3 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 [rose macro](#) (page 254) or on-demand in the [rose config-edit](#) (page 250) 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 [rose config-edit](#) (page 250) GUI or on the command line.

---

### 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.

### 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](#) (page 194) 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 [rose config-edit](#) (page 250) in the app directory. You should be able to navigate between the pages and view the help and description for the settings.

### **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](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 ( $=9.8 * \text{namelist:rocket=specific_impulse_s}$ ) 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.

### 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.

## 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 [Configuration Metadata](#) (page 206).

---

## 4.7.4 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.

### Warning

Macros should **only** be written if there is a genuine need that is not covered by [other metadata](#) (page 208) - make sure you are familiar with [Configuration Metadata](#) (page 206) 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 123) for details.

### 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 filesystem location. They can be invoked via the command line ([rose macro](#) (page 254)) or from within the *Metadata* menu in the config editor.

### 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` (page 194) 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` (page 207) 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 python
# -*- 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` (page 305). 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)^.*(?: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
```

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 python
# -*- 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.

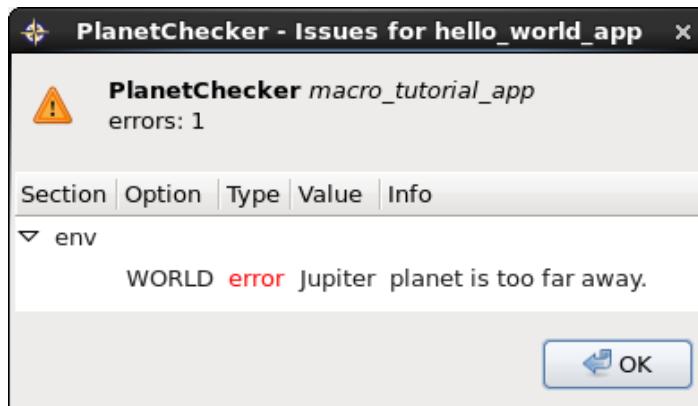
Run the config editor with the command:

```
rose edit
```

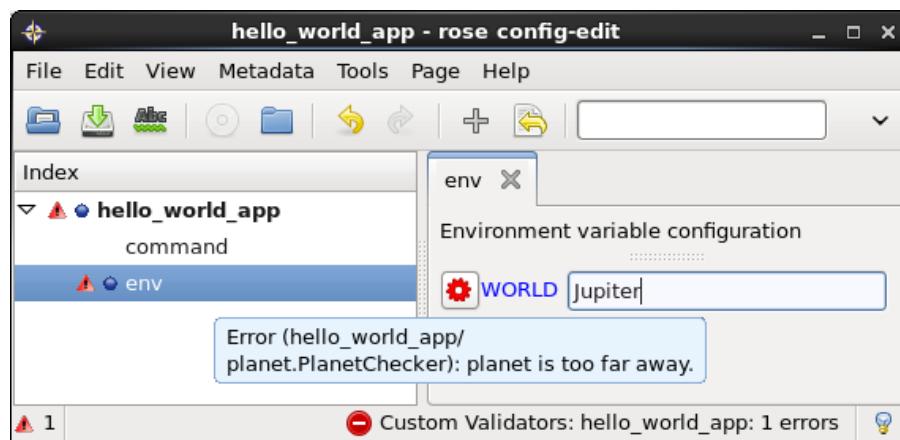
in the application directory. Navigate to the env page, and change the option env=WORLD to Jupiter.

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])
```

(continues on next page)

(continued from previous page)

```
# 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
```

(continues on next page)

(continued from previous page)

```

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)
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 `rose config-edit` (page 250) 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(GENERIC_HOROSCOPE_STATEMENTS))

```

(continues on next page)

(continued from previous page)

```

        generic_message = random.choice(
            self.GENERIC_HOROSCOPE_STATEMENTS)
    )

def get_planet_info(self, planet_name):
    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
```

## 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)]

```

(continues on next page)

(continued from previous page)

```
else:  
    new_planet = planet_name  
    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
```

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 [rose config-edit](#) (page 250). 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 [rose config-edit](#) (page 250) 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.

## 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.

### 4.7.5 Optional Configurations

Optional configurations are configuration files which can add or overwrite the default configuration. They can be used with [rose app-run](#) (page 245) for [Rose application configurations](#) and [rose suite-run](#) (page 263) for [Rose suite configurations](#).

## 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` (page 194) 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.
```

## 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 `rose app-run` (page 245) 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` (page 275) 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` (page 194) file.

## 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 are 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 *rose config* (page 247) 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* (page 275) environment variable are loaded before those specified using the *--opt-conf-key* command line option.

---

## Using Optional Configurations By Default

Optional configurations can be switched on by default using the *opts* setting.

Add the following line at the top of the *rose-app.conf* (page 194) 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* (page 275) environment variable instead.

## Other Optional Configurations

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

- Suites can have optional configurations that override *rose-suite.conf* (page 196) settings, controlled through *rose suite-run* (page 263). Optional suite configurations can be used either using the *--opt-conf-key* option with *rose suite-run* (page 263) or the *ROSE\_SUITE\_OPT\_CONF\_KEYS* (page 278) environment variable.
- Metadata configurations can also have optional configurations, typically included via the *opts* top-level setting.

### 4.7.6 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.

## Example

Create a new Rose suite configuration:

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

Create a blank `rose-suite.conf` (page 196) 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...'<br>  
    [[send_letter]]  
        env-script = eval `rose task-env`  
        script = """  
            sleep 5  
            echo 'Hello Bob' > $ROSE_DATA/letter.txt  
            sleep 10  
        """<br>  
  
    [[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
```

## 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` (page 194) file.

Edit the `rose-app.conf` (page 194) 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 [rose suite-run](#) (page 263).

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.

## 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](#) (page 194) 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.

## 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 [Application Configuration](#) (page 193).

---

## 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.

## 4.7.7 Rose Arch

`rose_arch` (page 289) 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.

---

### Example

Create a new Rose suite configuration:

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

Create a blank `rose-suite.conf` (page 196) 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_DATAAC/file_zip
            echo 'solo' >> $ROSE_DATAAC/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` (page 194) 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=$ROSE_DATA/
target-prefix=$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* (page 194) 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?

```

## 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 *rose suite-log* (page 262) and inspect the output of the make\_files, archive\_files\_rsync and archive\_files\_scp tasks.

## Results Of “Archiving”

Change to the \$ROSE\_DATA/ARCHVING/ directory of the suite i.e:

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

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 ARCHVING/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 diectory to ARCHVING/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) (<https://linux.die.net/man/1/rsync>) and [scp](https://www.lifewire.com/rcp-scp-ftp-commands-for-copying-files-3971107) (<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.

## Arch Settings

Some settings that can be used are described below. See the [rose\\_arch](#) (page 287) 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) (<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) (<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]` (page 289) 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]` (page 194) 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` (page 195).

**[file:TARGET]** This option allows the user to, for example, make the directory TARGET, e.g. `*[file:TARGET]mode=mkdir` (page 227).

For more information, see the [rose\\_arch](#) (page 287) documentation.

## 4.7.8 Rose Bunch

`rose_bunch` (page 291) is a built-in [Rose app](#) which allows multiple variants of a command to be run under a single job.

## 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 134) to change the way a command is to be run, we can instead use the built-in *rose\_bunch* (page 291) 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* (page 291) 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.

## 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* (page 196) 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* (page 194) 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](#) (page 291) task that calls multiple instances of the land command, supplying arguments to each instance from the class and passengers entries under [rose\\_bunch\[bunch-args\]](#) (page 292).

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](#) (page 196) files can be found) and run [rose suite-run](#) (page 263).

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 [rose suite-log](#) (page 262) 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:

task status	job status	cycle point	task name	job #	submit time	queue Δt	run Δt	job host	job batch	job logs
succeeded			lander	1 of 1	2 minutes ago	0.01	0.50	localhost	background[11216]	<a href="#">job</a> <a href="#">job-activity.log</a> <a href="#">job.err</a> <a href="#">job.out</a> <a href="#">job.status</a> <a href="#">bunch.*.err</a> <a href="#">bunch.*.out</a>

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`](#) (page 291) 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.

## 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`](#) (page 291) supports naming command instances via the [`rose\_bunch\[bunch\]names=`](#) (page 292) option.

Open your app config (under `app/lander/rose-app.conf`) and add the following line under the [`rose\_bunch\[bunch\]`](#) (page 291) 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.

## 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`](#) (page 291) 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=`](#) (page 291) configuration option of the [`rose\_bunch`](#) (page 291) app.

Open your app config (under `app/lander/rose-app.conf`) and add the following line to the [`rose\_bunch\[bunch\]`](#) (page 291) 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`](#) (page 291), the line will appear pretty quickly.

## Summary

In this tutorial we have learnt how to configure a `rose_bunch` (page 291) 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` (page 291) documentation. These include configuring how to proceed following failure of an individual command invocation (`rose_bunch[bunch] fail-mode=`(page 291)), automatically generating  $N$  command instances and enabling/disabling the app's incremental mode.

### 4.7.9 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.

## 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.

## Rose Stem

There are two components in Rose Stem:

`rose stem` (page 259) The command line tool which executes an appropriate suite.

`rose_ana` (page 286) 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.

### Running A Suite With `rose stem`

The `rose stem` command is essentially a wrapper to `rose suite-run` (page 263), 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.

## 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 `rose stem` (page 259) 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` (page 196) 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`.

## 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.

## 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.

### 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` (page 286) built-in application.

A Rose Ana `rose-app.conf` (page 194) 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]` (page 200) 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` (page 200) 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]` (page 200)
- `rose_ana[ana:config]` (page 286)

**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 151):

```
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:

<code>self.config</code>	A dictionary containing any Rose Ana configuration options.
<code>self.reporter</code>	A reference to the <code>rose.reporter.Reporter</code> instance used by the parent app (for printing to stderr/stdout).
<code>self.kgo_db</code>	A reference to the KGO database object created by the parent app (for adding entries to the database).
<code>self.popen</code>	A reference to the <code>rose.popen.RosePopener</code> 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 namespaces). 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.

---

## 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.` (page 200).

---

**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.

---

## Summary

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

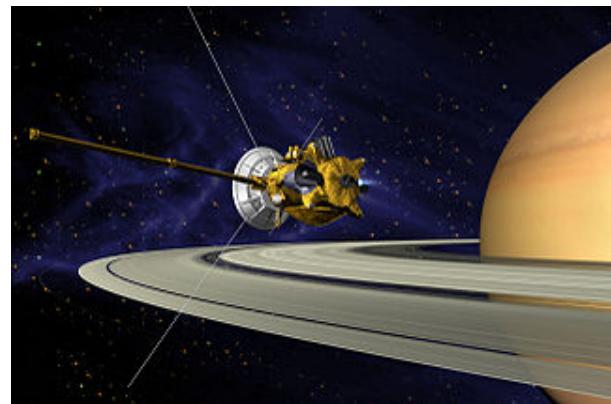
```
rose stem --group=groupname
```

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 152)

## Rose Stem Tutorial

**Warning:** Before proceeding you should already be familiar with the [Rose Stem](#) (page 147) 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.

## Getting Started

We will start the Rose Stem tutorial by setting up an [FCM](https://metomi.github.io/fcm/doc/) (<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) ([https://metomi.github.io/fcm/doc/user\\_guide/code\\_management.html#svn\\_basic\\_keywords](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 .
```

### `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.

## 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` (page 194) 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
```

## The fcm-make app

We now need to provide the instructions for `fcm_make` (page 284) to build the Fortran executable.

Create a new app called `fcm_make_spaceship` with an empty `rose-app.conf` (page 194) file.

Inside this app create a subdirectory called `file` and paste the following into the `fcm-make.cfg` file within that directory:

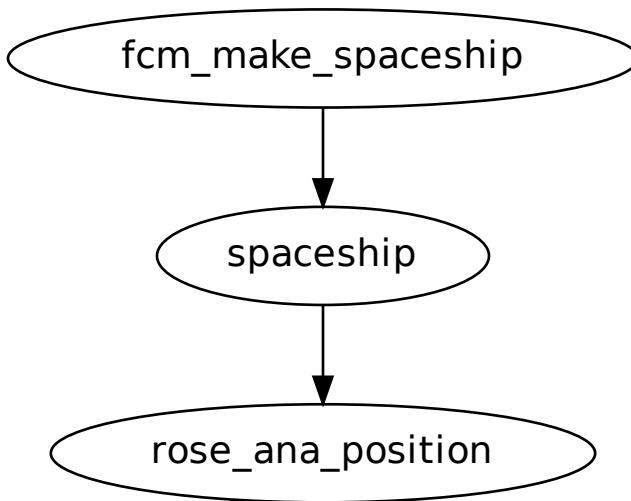
```
steps = build
build.source = $SOURCE_SPACESHIP/src
build.target{task} = link
```

The `$SOURCE_SPACESHIP` environment variable will be set using the Jinja2 variable of the same name which is provided by Rose Stem.

## 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* (page 34). 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.

### **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` (page 196) file, together with the default values of `RUN_NAMES` and `SOURCE_SPACESHIP`. Paste the following into your `rose-suite.conf` (page 196) 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.

### **The `rose_ana_position` app**

The final component is a `rose_ana` (page 286) 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` (page 194) file.

```
[ana:grepper.FilePattern(Check X position at each timestep)]
pattern='^s*Position:\s*(.*?)\s*,'
files=/home/user/spaceship/kgo.txt
=../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.

## 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` (page 194) 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.

## 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*).

## 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 Cyc server, and your home directory is not shared with the Cyc 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`.

## 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.

## 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.

## Automatic Options

It is possible to automatically add options to `rose stem` using the `rose.conf[rose-stem]automatic-options` (page 202) variable in the *Site And User Configuration* (page 198) 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.

### 4.7.10 Trigger

The `trigger` metadata item can be used to cut down the amount of irrelevant settings presented to the user in the `rose config-edit` (page 250) 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).

## 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` (page 194) file that looks like this:

```
[command]
default=order.exe

[env]
BUDGET=10
```

(continues on next page)

(continued from previous page)

```
[file:order.nl]
source=namelist:pizza_order namelist:side_order

[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` (page 207) 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 `rose config-edit` (page 250) 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'.

## Adding Triggers

Let's add some trigger information.

In the `rose-meta.conf` (page 207) 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 [Metadata Mini-Language](#) (page 219) 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.

## 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.

## 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.

## 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`.

## 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.

## Further Reading

For more information see *Configuration Metadata* (page 206).

### 4.7.11 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.

## Example

Create a new Rose application called garden:

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

Create within it a *rose-app.conf* (page 194) 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.

### `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 `rose app-upgrade` (page 246). 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
```

### 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` (page 194) 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` (page 194) file.

If you run Rose `rose app-upgrade` (page 246) with no arguments, you can see that you're using the latest version.

### 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:

- 
- [Configuration Metadata](#) (page 206)
  - [Rose Upgrade Macro API](#) (page 309)
- 

## 4.7.12 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 [rose app-upgrade](#) (page 246) (see the [Upgrading tutorial](#) (page 161) and the concepts in the reference material).

### 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.

### 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* (page 207) 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* (page 207) 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* (page 207) 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) ([https://en.wikipedia.org/wiki/Outrigger\\_canoe](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
outtrigger_tree_trunks=2
paddling_branches=1
```

It looks like we've added the inputs *misc\_branches*, *outtrigger\_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* (page 207) file that looks like this:

```
[namelist:materials=hollow_tree_trunks]
compulsory=true
values=1

[namelist:materials=misc_branches]
compulsory=true
range=4:
```

(continues on next page)

(continued from previous page)

```
[namelist:materials=paddling_branches]
compulsory=true
range=1:
type=integer

[namelist:materials=outrigger_tree_trunks]
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.

### **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 `rose app-upgrade` (page 246)), 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"
```

(continues on next page)

(continued from previous page)

```
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` (page 309), 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 Upgrade Macro API* (page 309) 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` (page 207) 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
```

(continues on next page)

(continued from previous page)

```
range=4:

[namelist:materials=outrigger_tree_trunks]
compulsory=true
values=2

[namelist:materials=paddling_branches]
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` (page 194) should now contain the new option, `l_rudder_branch`.

## More Complex Upgrade Macros

The *Rose Upgrade Macro API* (page 309) 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 our 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
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.

---

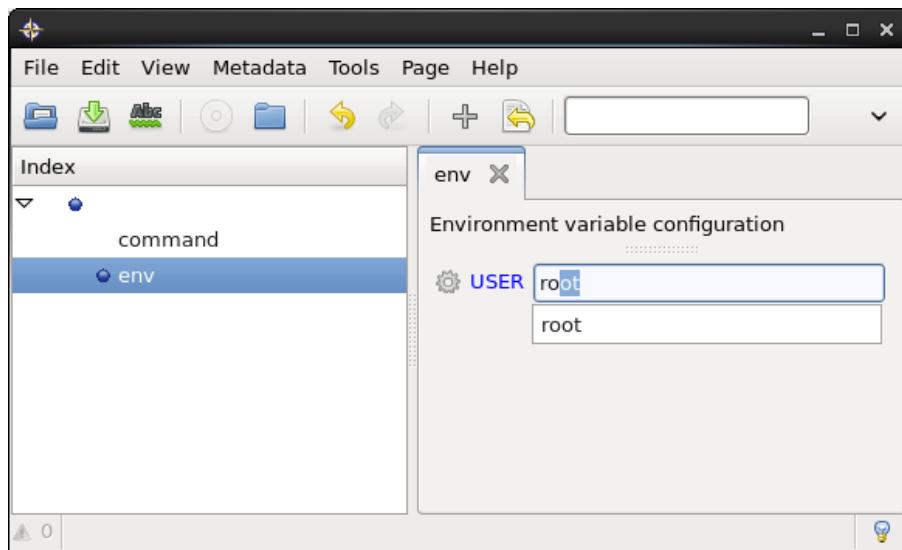
**Tip:** See also:

- [Rose Upgrade Macro API](#) (page 309)
  - [Rose Macro API](#) (page 303)
- 

### 4.7.13 Widget Development

The [rose config-edit](#) (page 250) GUI displays configurations using built-in widgets. For more complex requirements [rose config-edit](#) (page 250) 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.

## 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` (page 194) 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` (page 194) 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.

### Further Reading

For more information, see *Rose GTK library* (page 295) and the [PyGTK](http://www.pygtk.org/) (<http://www.pygtk.org/>) web page.



## CHEAT SHEET

This page outlines how to perform suite operations for “pure” *Cylc suites (the Cylc way)* and those using *Rose suite configurations (the Rose way)*.

### 5.1 Running/Interacting With Suites

#### 5.1.1 Starting Suites

The Cylc Way	The Rose way
<pre>cylc validate &lt;name&gt; cylc run &lt;name&gt;</pre>	<pre># run the suite in the current ↴ # directory: rose suite-run  # run a suite in another directory: rose suite-run -C &lt;path&gt;  # run using a custom name: rose suite-run --name &lt;name&gt;</pre>

#### 5.1.2 Stopping Suites

```
# Wait for running/submitted tasks to finish then shut down the suite:
cylc stop <name>

# Kill all running/submitted tasks then shut down the suite:
cylc stop <name> --kill

# Shut down the suite now leaving any running/submitted tasks behind.
# If the suite is restarted Cylc will "re-connect" with these jobs,
# continuing where it left off:
cylc stop <name> --now --now
```

#### 5.1.3 Restarting Suites (From Stopped)

Pick up a suite where it left off after a shutdown. Cylc will “re-connect” with any jobs from the previous run.

The CycL Way	The Rose Way
cyclc restart <name>	<pre># Restart the suite from the run # directory (recommended): rose suite-restart</pre> <pre># Re-install the suite from the suite # directory then restart: rose suite-run --restart</pre>

### 5.1.4 Restarting Suites (From Running)

*This might be needed, for instance, to upgrade a running suite to a newer version of CycL.*

Stop a suite leaving all running/submitted jobs unchanged, then restart the suite without making any changes to the [run directory](#). CycL will “re-connect” with any jobs from the previous run.

The CycL Way	The Rose Way
cyclc stop <name> --now --now cyclc restart <name>	cyclc stop <name> --now --now rose suite-restart

### 5.1.5 Reloading Suites

Change the configuration of a running suite.

The CycL Way	The Rose Way
cyclc reload <name>	<pre># Re-install the suite run directory ↴ # then # perform `cyclc reload`: rose suite-run --reload</pre>

## 5.2 Scanning/Inspecting Suites

### 5.2.1 List Running Suites

```
# On the command line:
cyclc scan

# Via a GUI:
cyclc gscan
```

### 5.2.2 Visualise A Running Suite

```
cyclc gui <name>
```

### 5.2.3 Visualise A Suite's Graph

The CycL Way	The Rose Way
# No special steps required.	# Only if the suite is not running: rose suite-run -l

```
cyclc graph <name>
```

### 5.2.4 View A Suite's suite.rc Configuration

The CycL Way	The Rose Way
# No special steps required.	# Only if the suite is not running: rose suite-run -l

```
cyclc get-config --sparse <name or path-to-suite>

# View the "full" configuration with defaults included:
cyclc get-config <name or path-to-suite>

# View a specific configuration item (e.g. "[scheduling]initial cycle point"):
cyclc get-config <name or path-to-suite> -i <item>
```



---

**CHAPTER  
SIX**

---

## **GLOSSARY**

**application directory** The application directory is the folder in which the `rose-app.conf` (page 194) file is located in a *Rose application configuration*.

**batch system** A batch system or job scheduler is a system for submitting *jobs* onto a compute platform.

See also:

- [Wikipedia \(job scheduler\)](https://en.wikipedia.org/wiki/Job_scheduler) ([https://en.wikipedia.org/wiki/Job\\_scheduler](https://en.wikipedia.org/wiki/Job_scheduler))
- *directive*

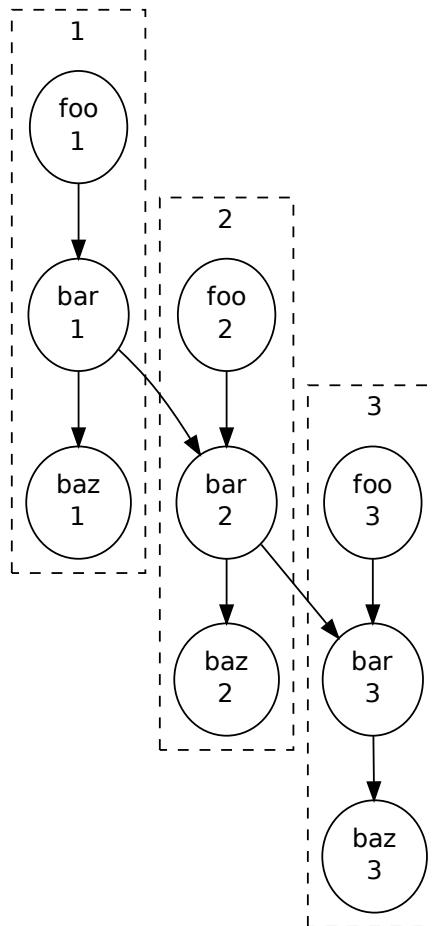
**cold start** A cold start is one in which the *suite starts* from the *initial cycle point*. This is the default behaviour of `cylc run`.

See also:

- *warm start*

**cycle** In a *cycling suite* one cycle is one repetition of the workflow.

For example, in the following workflow each dotted box represents a cycle and the *tasks* within it are the *tasks* belonging to that cycle. The numbers (i.e. 1, 2, 3) are the *cycle points*.



**cycle point** A cycle point is the unique label given to a particular *cycle*. If the *suite* is using *integer cycling* then the cycle points will be numbers e.g. 1, 2, 3, etc. If the *suite* is using *datetime cycling* then the labels will be *ISO8601* datetimes e.g. 2000-01-01T00:00Z.

See also:

- *initial cycle point*
- *final cycle point*

**cycling** A cycling *suite* is one in which the workflow repeats.

See also:

- *cycle*
- *cycle point*

**datetime cycling** A datetime cycling is the default for a *cycling suite*. When using datetime cycling *cycle points* will be *ISO8601 datetimes* e.g. 2000-01-01T00:00Z and *ISO8601 recurrences* can be used e.g. P3D means every third day.

See also:

- *CycL tutorial* (page 31)

**dependency** A dependency is a relationship between two *tasks* which describes a constraint on one.

For example the dependency `foo => bar` means that the `task` `bar` is *dependent* on the task `foo`. This means that the task `bar` will only run once the task `foo` has successfully completed.

See also:

- `task trigger`

**directive** Directives are used by *batch systems* to determine what a *job's* requirements are, e.g. how much memory it requires.

Directives are set in the `suite.rc` file in the `[runtime]` section (`[runtime] [<task-name>] [directives]`).

See also:

- `batch system`

**family** In Cyc a family is a collection of `tasks` which share a common configuration and which can be referred to collectively in the `graph`.

By convention families are named in upper case with the exception of the special `root` family from which all tasks inherit.

See also:

- *Cyclc tutorial* (page 53)
- `family inheritance`
- `family trigger`

**family inheritance** A `task` can be “added” to a `family` by “inheriting” from it.

For example the `task` `task` “belongs” to the `family` `FAMILY` in the following snippet:

```
[runtime]
  [[FAMILY]]
    [[[environment]]]
      FOO = foo
  [[task]]
    inherit = FAMILY
```

A task can inherit from multiple families by writing a comma-separated list e.g:

```
inherit = foo, bar, baz
```

See also:

- *Cyclc User Guide* (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>)
- `family`
- `family trigger`

**family trigger** `Tasks` which “belong” to (`inherit` from) a `family` can be referred to collectively in the `graph` using a family trigger.

A family trigger is written using the name of the family followed by a special qualifier i.e. `FAMILY_NAME:qualifier`. The most commonly used qualifiers are:

**succeed-all** The dependency will only be met when **all** of the tasks in the family have **succeeded**.

**succeed-any** The dependency will be met as soon as **any one** of the tasks in the family has **succeeded**.

**finish-all** The dependency will only be met once **all** of the tasks in the family have **finished** (either succeeded or failed).

See also:

- *Cyclc User Guide* (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>)
- `family`

- *task trigger*
- *dependency*
- *Family Trigger Tutorial* (page 69)

**final cycle point** In a *cycling suite* the final cycle point is the point at which cycling ends.

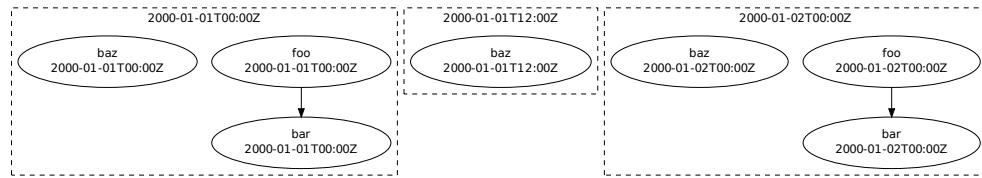
If the final cycle point were 2001 then the final cycle would be no later than the 1st of January 2001.

See also:

- *cycle point*
- *initial cycle point*

**graph** The graph of a *suite* refers to the *graph strings* contained within the [scheduling] [dependencies] section. For example the following is, collectively, a graph:

```
[P1D]
graph = foo => bar
[PT12H]
graph = baz
```



**graph string** A graph string is a collection of dependencies which are placed under a *graph* section in the *suite.rc* file. E.G:

```
foo => bar => baz & pub => qux
pub => bool
```

**initial cycle point** In a *cycling suite* the initial cycle point is the point from which cycling begins.

If the initial cycle point were 2000 then the first cycle would be on the 1st of January 2000.

See also:

- *cycle point*
- *final cycle point*

**integer cycling** An integer cycling suite is a *cycling suite* which has been configured to use integer cycling.

When a suite uses integer cycling integer *recurrences* may be used in the *graph*, e.g. P3 means every third cycle. This is configured by setting [scheduling]cycling mode = integer in the *suite.rc* file.

See also:

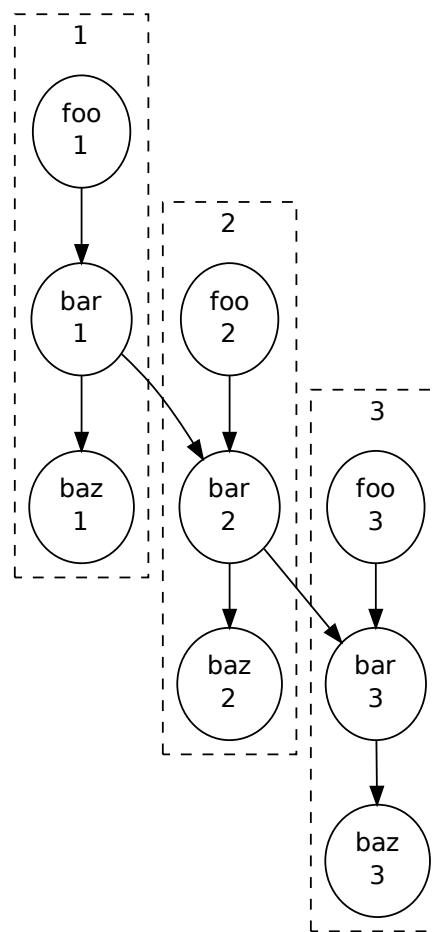
- *Cycle tutorial* (page 22)

**inter-cycle dependency** In a *cycling suite* an inter-cycle dependency is a *dependency* between two tasks in different cycles.

For example in the following suite the task *bar* is dependent on its previous occurrence:

```
[scheduling]
initial cycle point = 1
cycling mode = integer
[[dependencies]]
[[[P1]]]
graph = """
    foo => bar => baz
    bar[-P1] => bar
"""

```



**ISO8601** ISO8601 is an international standard for writing dates and times which is used in Cylc with *datetime cycling*.

See also:

- *ISO8601 datetime*
- *recurrence*
- Wikipedia (ISO8601) ([https://en.wikipedia.org/wiki/ISO\\_8601](https://en.wikipedia.org/wiki/ISO_8601))
- International Organisation For Standardisation (<https://www.iso.org/iso-8601-date-and-time-format.html>)

- a summary of the international standard date and time notation (<http://www.cl.cam.ac.uk/%7Emgk25/iso-time.html>)

**ISO8601 datetime** A date-time written in the ISO8601 format, e.g:

- 2000-01-01T00:00Z: midnight on the 1st of January 2000

See also:

- *Cyclc tutorial* (page 31)
- *ISO8601*

**ISO8601 duration** A duration written in the ISO8601 format e.g:

- PT1H30M: one hour and thirty minutes.

See also:

- *Cyclc tutorial* (page 32)
- *ISO8601*

**job** A job is the realisation of a *task* consisting of a file called the *job script* which is executed when the job “runs”.

See also:

- *task*
- *job script*

**job host** The job host is the compute platform that a *job* runs on. For example `some-host` would be the job host for the task `some-task` in the following suite:

```
[runtime]
  [[some-task]]
    [[[remote]]]
      host = some-host
```

## job log

**job log directory** When Cyclc executes a *job*, `stdout` and `stderr` are redirected to the `job.out` and `job.err` files which are stored in the job log directory.

The job log directory lies within the *run directory*:

```
<run directory>/log/job/<cycle>/<task-name>/<submission-no>
```

Other files stored in the job log directory:

- *job*: the *job script*.
- *job-activity.log*: a log file containing details of the *job's* progress.
- *job.status*: a file holding Cyclc's most up-to-date understanding of the *job's* present status.

**job script** A job script is the file containing a bash script which is executed when a *job* runs. A task's job script can be found in the *job log directory*.

See also:

- *task*
- *job*
- *job submission number*

**job submission number** Cyclc may run multiple *jobs* per *task* (e.g. if the task failed and was re-tried). Each time Cyclc runs a *job* it is assigned a submission number. The submission number starts at 1, incrementing with each submission.

See also:

- *job*
- *job script*

## metadata

**Rose metadata** Rose metadata provides information about settings in *Rose application configurations* and *Rose suite configurations*. This information is stored in a *rose-meta.conf* (page 207) file in a *meta/* directory alongside the configuration it applies to.

This information can include:

- Documentation and help text, e.g. *rose-meta.conf[SETTING]title* (page 216) provides a short title to describe a setting.
- Information about permitted values for the setting, e.g. *rose-meta.conf[SETTING]type* (page 208) can be used to specify the data type a setting requires (integer, string, boolean, etc).
- Settings affecting how the configurations are displayed in *rose config-edit* (page 250) (e.g. *rose-meta.conf[SETTING]sort-key* (page 208)).
- Metadata which defines how settings should behave in different states (e.g. *rose-meta.conf[SETTING]trigger* (page 213)).

This information is used for:

- Presentation and validation in the *rose config-edit* (page 250) GUI.
- Validation using the *rose macro* (page 254) command.

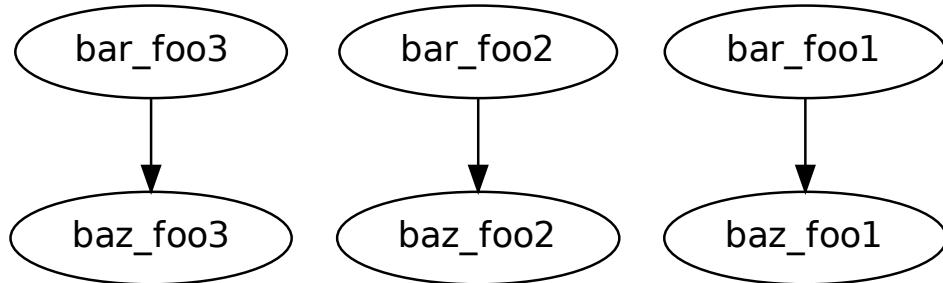
Metadata does not affect the running of an *application* or *Cylc suite*.

See also:

- *Configuration Metadata* (page 206)

**parameterisation** Parameterisation is a way to consolidate configuration in the Cylc suite.rc file by implicitly looping over a set of pre-defined variables e.g:

```
[cylc]
  [[parameters]]
    foo = 1..3
[scheduling]
  [[dependencies]]
    graph = bar<foo> => baz<foo>
```



See also:

- *Cylc tutorial* (page 60)

**qualifier** A qualifier is used to determine the *task state* to which a *dependency* relates.

See also:

- *Cylc tutorial* (page 38)
- *task state*

**recurrence** A recurrence is a repeating sequence which may be used to define a *cycling suite*. Recurrences determine how often something repeats and take one of two forms depending on whether the *suite* is configured to use *integer cycling* or *datetime cycling*.

See also:

- *integer cycling*
- *datetime cycling*

**reload** Any changes made to the `suite.rc` file whilst the suite is running will not have any effect until the suite is either:

- *Shutdown* and *rerun*
- *Shutdown* and *restarted*
- “Reloaded”

Reloading does not require the suite to be *shutdown*. When a suite is reloaded any currently “active” *tasks* will continue with their “pre-reload” configuration, whilst new tasks will use the new configuration.

Reloading changes is safe providing they don’t affect the *suite’s graph*. Changes to the graph have certain caveats attached, see the *Cylc User Guide* (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>) for details.

See also:

- *Reloading Suites* (page 176)
- *Cylc User Guide* (<http://cylc.github.io/cylc/documentation.html#the-cylc-user-guide>)

**restart** When a *stopped suite* is “restarted” Cylc will pick up where it left off. Cylc will detect any *jobs* which have changed state (e.g. succeeded) during the period in which the *suite* was *shutdown*.

See also:

- *Restarting Suites (From Stopped)* (page 175)
- *start*
- *Stop*
- *Reload*

## Rose app

### Rose application

**Rose application configuration** A Rose application configuration (or Rose app) is a directory containing a `rose-app.conf` (page 194) file along with some other optional files and directories.

An application can configure:

- The command to run (`rose-app.conf[command]` (page 194)).
- Any environment variables to provide it with (`rose-app.conf[env]` (page 194))
- Input files e.g. namelists (`rose-app.conf[nameList:NAME]` (page 195))
- Metadata for the application (`rose-meta.conf` (page 207)).

See also:

- *Application Configuration* (page 193)

**Rose built-in application** A Rose built-in application is a generic *Rose application* providing common functionality which is provided in the Rose installation.

See also:

- [Rose Built-In Applications](#) (page 283)

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

The two types of Rose configuration relevant to Cylc suites are:

- [Rose application configuration](#)
- [Rose suite configuration](#)

See also:

- [Rose Configuration Format](#) (page 223)
- [Rose Configuration Tutorial](#) (page 97)
- [Optional Configuration Tutorial](#) (page 134)

**Rose suite configuration** A Rose suite configuration is a `rose-suite.conf` (page 196) file along with other optional files and directories which configure the way in which a *Cylc suite* is run. E.g:

- Jinja2 variables to be passed into the `suite.rc` file ([rose-suite.conf\[jinja2:suite.rc\]](#) (page 197)).
- Environment variables to be provided to `cylc run` ([rose-suite.conf\[env\]](#) (page 196)).
- Installation configuration (e.g. `rose-suite.conf/root-dir` (page 197), `rose-suite.conf[file:NAME]` (page 197)).

See also:

- [Suite Configuration](#) (page 196)

**Rosie Suite** A Rosie suite is a *Rose suite configuration* which is managed using the Rosie system.

When a suite is managed using Rosie:

- The *suite directory* is added to version control.
- The suite is registered in a database.

See also:

- [Rosie Tutorial](#) (page 113)

**run directory** When a *suite* is run a directory is created for all of the files generated whilst the suite is running. This is called the run directory and typically resides in the `cylc-run` directory:

`~/cylc-run/<suite-name>`

---

**Note:** If a suite is written in the `cylc-run` directory the run directory is also the *suite directory*.

---

The `run directory` can be accessed by a running suite using the environment variable `CYLC_SUITE_RUN_DIR`.

See also:

- [suite directory](#)
- [Suite Directory Vs Run Directory](#) (page 106)
- [work directory](#)
- [share directory](#)
- [job log directory](#)

**share directory** The share directory resides within a suite's *run directory*. It serves the purpose of providing a storage place for any files which need to be shared between different tasks.

`<run directory>/share`

The location of the share directory can be accessed by a *job* via the environment variable CYLC\_SUITE\_SHARE\_DIR.

In cycling suites files are typically stored in cycle sub-directories.

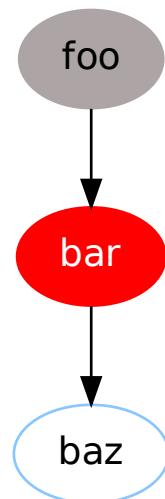
See also:

- *run directory*
- *work directory*

### stalled suite

**stalled state** If Cylc is unable to proceed running a workflow due to unmet dependencies the suite is said to be *stalled*.

This usually happens because of a task failure as in the following diagram:



In this example the task bar has failed meaning that baz is unable to run as its dependency (bar:succeed) has not been met.

When a Cylc detects that a suite has stalled an email will be sent to the user. Human interaction is required to escape a stalled state.

### start

**startup** When a *suite* starts the Cylc *suite server program* is run. This program controls the suite and is what we refer to as “running”.

- A *Cylc suite* is started using cylc run.
- A *Rose suite configuration* (or *Rosie Suite*) is started using rose suite-run (page 263).

A suite start can be either *cold* or *warm* (cold by default).

See also:

- *Running/Interacting With Suites* (page 175)

- *suite server program*
- *warm start*
- *cold start*
- *shutdown*
- *restart*
- *reload*

**stop**

**shutdown** When a *suite* is shutdown the *suite server program* is stopped. This means that no further *jobs* will be submitted.

By default Cycl waits for any submitted or running *jobs* to complete (either succeed or fail) before shutting down.

See also:

- *Stopping Suites* (page 175)
- *start*
- *restart*
- *reload*

**suite**

**Cycl suite** A Cycl suite is a directory containing a `suite.rc` file which contains *graphing* representing a workflow.

See also:

- *Relationship between Cycl suites, Rose suite configurations and Rosie suites* (page 118)

**suite directory** The suite directory contains all of the configuration for a suite (e.g. the `suite.rc` file and for Rose suites the `rose-suite.conf` (page 196) file).

This is the directory which is registered using `cyclc reg` or, for Rose suites, it is the one in which the `rose suite-run` (page 263) command is executed.

---

**Note:** If a suite is written in the `cyclc-run` directory the suite directory is also the *run directory*.

---

See also:

- *run directory*
- *Rose suite installation diagram* (page 107)

**suite log**

**suite log directory** A Cycl suite logs events and other information to the suite log files when it runs. There are three log files:

- `out` - the `stdout` of the suite.
- `err` - the `stderr` of the suite, which may contain useful debugging information in the event of any error(s).
- `log` - a log of suite events, consisting of information about user interaction.

The suite log directory lies within the *run directory*:

```
<run directory>/log/suite
```

**suite server program** When we say that a *suite* is “running” we mean that the cylc suite server program is running.

The suite server program is responsible for running the suite. It submits *jobs*, monitors their status and maintains the suite state.

By default a suite server program is a *daemon* ([https://en.wikipedia.org/wiki/Daemon\\_\(computing\)](https://en.wikipedia.org/wiki/Daemon_(computing))) meaning that it runs in the background (potentially on another host).

**task** A task represents an activity in a workflow. It is a specification of that activity consisting of the script or executable to run and certain details of the environment it is run in.

The task specification is used to create a *job* which is executed on behalf of the task.

Tasks submit *jobs* and therefore each *job* belongs to one task. Each task can submit multiple *jobs*.

See also:

- *job*
- *job script*

**task state** During a *task’s* life it will proceed through various states. These include:

- waiting
- running
- succeeded

See also:

- *Cylc tutorial* (page 40)
- *task*
- *job*
- *qualifier*

**task trigger** *Dependency* relationships can be thought of the other way around as “triggers”.

For example the dependency `foo => bar` could be described in two ways:

- “`bar` is dependent on `foo`”
- “`foo` is triggered by `bar`”

In practice a trigger is the left-hand side of a dependency (`foo` in this example).

See also:

- *dependency*
- *qualifier*
- *family trigger*

**wall-clock time** In a Cylc suite the wall-clock time refers to the actual time (in the real world).

See also:

- *datetime cycling*
- *Clock Trigger Tutorial* (page 65)

**warm start** In a *cycling suite* a warm start is one in which the *suite starts* from a *cycle point* after the *initial cycle point*. Tasks in cycles before this point are assumed to have succeeded.

See also:

- *cold start*

**work directory** When Cylc executes a *job* it does so inside the *job’s* working directory. This directory is created by Cylc and lies within the directory tree inside the relevant suite’s *run directory*.

```
<run_directory>/work/<cycle>/<task-name>
```

The location of the work directory can be accessed by a *job* via the environment variable CYLC\_TASK\_WORK\_DIR.

Any files installed by *Rose apps* will be placed within this directory.

See also:

- *run directory*
- *share directory*
- *Rose suite installation diagram* (page 107)



## ROSE CONFIGURATION

Rose is configured by a collection of configuration files which all use the same *Rose Configuration Format* (page 223). They are used for:

### *Application Configuration* (page 193)

- *rose-app.conf* (page 194)

### *Suite Configuration* (page 196)

- *rose-suite.conf* (page 196)
- *rose-suite.info* (page 197)

### *Site And User Configuration* (page 198)

- *rose.conf* (page 198)

### *Configuration Metadata* (page 206)

- *rose-meta.conf* (page 207)

## 7.1 Application Configuration

The configuration of an application is represented by a directory. It may contain the following:

- *rose-app.conf* (page 194) - a compulsory configuration file in the modified INI format. It contains the following information:
  - the command(s) to run.
  - the metadata type for the application.
  - the list of environment variables.
  - other configurations that can be represented in un-ordered key=value pairs, e.g. Fortran namelists.
- file/ directory: other input files, e.g.:
  - FCM configuration files (requires ordering of key=value pairs).
  - STASH files.
  - other configuration files that require more than section=key=value.

Files in this directory are copied to the working directory in run time.

---

**Note:** If there is a clash between a [file:\*] section and a file under file/, the setting in the [file:\*] section takes precedence. E.g. Suppose we have a file file/hello.txt. In the absence of [file:hello.txt], it will copy file/hello.txt to \$PWD/hello.txt in run time. However, if we have a [file:hello.txt] section and a source=SOURCE setting, then it will install the file from SOURCE instead. If we have [!file:hello.txt], then the file will not be installed at all.

---

- bin/ directory for e.g. scripts and executables used by the application at run time. If a bin/ exists in the application configuration, it will be prepended to the PATH environment variable at run time.
- meta/ directory for the metadata of the application.
- opt/ directory (see [Optional Configuration](#) (page 225)).

E.g. The application configuration directory may look like:

```
./bin/  
./rose-app.conf  
./file/file1  
./file/file2  
./meta/rose-meta.conf  
./opt/rose-app-extra1.conf  
./opt/rose-app-extra2.conf  
...
```

## Rose Configuration rose-app.conf

### **Config file-install-root**

Root level setting. Specify the root directory to install file targets that are specified with a relative path.

### **Config meta**

Root level setting. Specify the configuration metadata for the application. This is ignored by the application runner, but may be used by other Rose utilities, such as [rose config-edit](#) (page 250). It can be used to specify the application type.

### **Config mode**

Root level setting. Specify the name of a built-in application, instead of running a command specified in the [\[command\]](#) (page 194) section.

See also [Rose Built-In Applications](#) (page 283).

### **Config [command]**

Specify the command(s) to run.

#### **Config default= COMMAND**

**Compulsory** True

Specify the default command to run.

#### **Config ALTERNATE= COMMAND**

Specify an alternate command referred to by the name ALTERNATE which can be selected at runtime.

See the [Command Keys](#) (page 120) tutorial.

### **Config [env]**

Specify environment variables to be provided to the [\[command\]](#) (page 194) at runtime.

The usual \$NAME or \${NAME} syntax can be used in values to reference environment variables that are already defined when the application runner is invoked. However, it is unsafe to reference other environment variables defined in this section.

If the value of an environment variable setting begins with a tilde ~, all of the characters preceding the first slash / are considered a *tilde-prefix*. Where possible, a tilde-prefix is replaced with the home directory associated with the specified login name at run time.

#### **Config KEY= VALUE**

Define an environment variable KEY with the value VALUE.

#### **Config UNDEF**

A special variable that is always undefined at run time.

Reference to it will cause a failure at run time. It can be used to indicate that a value must be overridden at run time.

**Config [etc]**

Specify misc. settings.

---

**Tip:** Currently, only UM defs for science sections are thought to require this section.

---

**Config [file:TARGET]**

Specify a file/directory to be installed. TARGET should be a path relative to the run time \$PWD or STDIN.

E.g. file:app/APP=source=LOCATION.

For a list of configuration options see [\\*\[file:TARGET\]](#) (page 227) for details.

**Config [namelist:NAME]**

Specify a Fortran namelist with the group name called NAME, which can be referred to by a [\\*\[file:TARGET\]source](#) (page 227) setting of a file.

**Config KEY= VALUE**

Define a new namelist setting KEY set to VALUE exactly like a Fortran namelist, but without the trailing comma.

Namelists can be grouped in two ways:

## 1. [namelist:NAME (SORT-INDEX) ]

- This allows different namelist files to have namelists with the same group name. These will all inherit the same group configuration metadata (from [namelist:NAME]).
- This allows the source setting of a file to refer to all [namelist:NAME (SORT-INDEX)] as namelist:NAME(:), and the namelist groups will be sorted alphanumerically by the SORT-INDEX.

## 2. [namelist:NAME {CATEGORY} ]

- This allows the same namelist to have different usage and configuration metadata according to its category. Namelists will inherit configuration metadata from their basic group [namelist:NAME] as well as from their specific category [namelist:NAME {CATEGORY}].

These groupings can be combined: [namelist:NAME {CATEGORY} (SORT-INDEX) ]

**Config [poll]**

Specify prerequisites to poll for before running the actual application. Three types of tests can be performed:

**Config all-files**

A list of space delimited list of file paths. This test passes only if all file paths in the list exist.

**Config any-files**

A list of space delimited list of file paths. This test passes if any file path in the list exists.

**Config test**

A shell command. This test passes if the command returns a 0 (zero) return code.

Normally, the [all-files](#) (page 195) and [any-files](#) (page 195) tests both test for the existence of file paths.

**Config file-test**

If [test](#) (page 195) is not enough, e.g. you want to test for the existence of a string in each file, you can specify a [file-test](#) (page 195) to do a grep. E.g.:

```
all-files=file1 file2
file-test=test -e {} && grep -q 'hello' {}
```

At runtime, any {} pattern in the above would be replaced with the name of the file. The above make sure that both file1 and file2 exist and that they both contain the string hello.

### Config delays

The above tests will only be performed once when the application runner starts. If a list of *delays* (page 195) are added, the tests will be performed a number of times with delays between them. If the prerequisites are still not met after the number of delays, the application runner will fail with a time out. The list is a comma-separated list. The syntax looks like  $[n*] [DURATION]$ , where DURATION is an *ISO8601 duration* (page 32) such as PT5S (5 seconds) or PT10M (10 minutes), and n is an optional number of times to repeat it. E.g.:

```
# Default
delays=0

# Poll 1 minute after the runner begins, repeat every minute 10 times
delays=10*PT1M

# Poll when runner begins,
# repeat every 10 seconds 6 times,
# repeat every minute 60 times,
# repeat once after 1 hour
delays=0,6*PT10S,60*PT1M,PT1H
```

## 7.2 Suite Configuration

The configuration and functionality of a suite will usually be covered by the use of *Cylc* (<http://cylc.github.io/cylc/>). In which case, most of the suite configuration will live in the *Cylc suite.rc* file. Otherwise, a suite is just a directory of files.

A suite directory may contain the following:

- A file called *rose-suite.conf* (page 196), a configuration file in the modified INI format described above. It stores the information on how to install the suite. See *rose-suite.conf* (page 196) for detail.
- A file called *rose-suite.info* (page 197), a configuration file in the modified INI format described above. It describes the suite's purpose and identity, e.g. the title, the description, the owner, the access control list, and other information. Apart from a few standard fields, a suite is free to store any information in this file. See *rose-suite.info* (page 197) for detail.
- An *app/* directory of application configurations used by the suite.
- A *bin/* directory of scripts and utilities used by the suite.
- An *etc/* directory of other configurations and resources used the suite. E.g. *fcm\_make* (page 284) configurations.
- A *meta/* directory containing the suite's configuration metadata.
- *opt/* directory. For detail, see *Optional Configuration* (page 225).
- Other items, as long as they do not clash with the scheduler's working directories. E.g. for a Cylc suite, *log\*/*, *share/*, *state/* and *work/* should be avoided.

### Rose Configuration *rose-suite.conf*

The suite install configuration file *rose-suite.conf* (page 196) should contain the information on how to install the suite.

#### Config [env]

Specify the environment variables to export to the suite daemon. The usual \$NAME or \${NAME} syntax can be used in values to reference environment variables that are already defined before the suite runner is invoked. However, it is unsafe to reference other environment variables defined in this section. If the value of an environment variable setting begins with a tilde ~, all of the characters preceding the first slash / are considered a *tilde-prefix*. Where possible, a tilde-prefix is replaced with the home directory associated with the specified login name at run time.

**Config KEY= VALUE**

Define an environment variable KEY with the value VALUE.

**Config ROSE\_VERSION= ROSE\_VERSION\_NUMBER**

If specified, the version of Rose that starts the suite run must match the specified version.

**Config CYLC\_VERSION= CYLC\_VERSION\_NUMBER**

If specified for a Cylc suite, the Rose suite runner will attempt to use this version of cylc.

**Config [jinja2:suite.rc]****Config KEY= VALUE**

Define a [Jinja2](http://jinja.pocoo.org/) (<http://jinja.pocoo.org/>) variable KEY with the value VALUE for use in the suite.rc file.

The assignment will be inserted after the #!jinja2 line of the installed suite.rc file.

**Config [file:NAME]**

Specify a file/directory to be installed. NAME should be a path relative to the run time \$PWD.

E.g. file:app/APP=source=LOCATION.

For a list of configuration options and details on each see [\\*\[file:TARGET\]](#) (page 227).

**Config meta**

Specify the configuration metadata for the suite. The section may be used by various Rose utilities, such as the config editor GUI. It can be used to specify the suite type.

**Config root-dir= LIST**

A new line delimited list of PATTERN=DIR pairs. The PATTERN should be a glob-like pattern for matching a host name. The DIR should be the root directory to install a suite run directory. E.g.:

```
root-dir=hpc*=$WORKDIR
      =*=$DATADIR
```

In this example, [rose suite-run](#) (page 263) of a suite with name \$NAME will create ~/cylc-run/\$NAME as a symbolic link to \$DATADIR/cylc-run/\$NAME/ on any machine, except those with their hostnames matching hpc\*. In which case, it will create ~/cylc-run/\$NAME as a symbolic link to \$WORKDIR/cylc-run/\$NAME/.

**Config root-dir{share}= LIST**

A new line delimited list of PATTERN=DIR pairs. The PATTERN should be a glob-like pattern for matching a host name. The DIR should be the root directory where the suite's share/ directory should be created.

**Config root-dir{share/cycle}= LIST**

A new line delimited list of PATTERN=DIR pairs. The PATTERN should be a glob-like pattern for matching a host name. The DIR should be the root directory where the suite's share/cycle/ directory should be created.

**Config root-dir{work}= LIST**

A new line delimited list of PATTERN=DIR pairs. The PATTERN should be a glob-like pattern for matching a host name. The DIR should be the root directory where the suite's work/ directory for tasks should be created.

**Config root-dir-share= LIST**

Deprecated since version 2015.04: Equivalent to [root-dir{share}=LIST](#) (page 197).

**Config root-dir-work= LIST**

Deprecated since version 2015.04: Equivalent to [root-dir{work}=LIST](#) (page 197).

**Rose Configuration rose-suite.info**

The suite information file [rose-suite.info](#) (page 197) should contain the information on identify and the purpose of the suite. It has no sections, only KEY=VALUE pairs. The owner, project and title settings are compulsory. Otherwise, any KEY=VALUE pairs can appear in this file. If the name of a KEY

ends with `-list`, the value is expected to be a space-delimited list. The following keys are known to have special meanings:

**Config owner**

Specify the user ID of the owner of the suite. The owner has full commit access to the suite. Only the owner can delete the suite, pass the suite's ownership to someone else or change the `access-list` (page 198).

**Config project**

Specify the name of the project associated with the suite.

**Config title**

Specify a short title of the suite.

**Config access-list**

Specify a list of users with commit access to trunk of the suite. A `*` in the list means that anyone can commit to the trunk of the suite. Setting this blank or omitting the setting means that nobody apart from the owner can commit to the trunk. Only the suite owner can change the access list.

**Config description**

Specify a long description of the suite.

**Config sub-project**

Specify a sub-division of `project` (page 198), if applicable.

## 7.3 Site And User Configuration

Aspects of some Rose utilities can be configured per installation via the site configuration file and per user via the user configuration file. Any configuration in the site configuration overrides the default, and any configuration in the user configuration overrides the site configuration and the default. Rose expects these files to be in the modified INI format described in *Rose Configuration Format* (page 223). Rose utilities search for its site configuration at `$ROSE_HOME/etc/rose.conf` where `$ROSE_HOME/bin/rose` is the location of the `rose` command, and they search for the user configuration at `$HOME/.metomi/rose.conf` where `$HOME` is the home directory of the current user.

You can also override many internal constants of `rose config-edit` (page 250) and `rosie go`. To change the keyboard shortcut of the Find Next action in the config editor to F3, put the following lines in your user config file, and the setting will apply the next time you run `rose config-edit` (page 250):

```
[rose-config-edit]
accel-find-next=F3
```

### Rose Configuration `rose.conf`

The `rose.conf` (page 198) file can be installed in:

- `$ROSE_HOME/etc/` for *site* configuration.
- `$HOME/.metomi/rose/` for *user* configuration.

```
Config checksum-method= md5|sha1|...
    default md5
```

Default method for checksum calculation. Values can be any algorithm available from Python's `hashlib`.

```
Config meta-path= DIR1[:DIR2[:...]]
```

Paths to locate configuration metadata e.g. `meta-path=/opt/rose-meta`.

```
Config rose-doc= http://metomi.github.io/rose/doc/
    default file://${ROSE_HOME}/doc/
```

URL to Rose documentation.

```

Config [external]
    Configuration of external commands.

Config diff_tool= COMMAND
    default diff -u
        Launch diff tool.

Config editor= COMMAND
    default vi
        Launch text editor.

Config fs_browser= COMMAND
    default nautilus
        Launch graphical file system browser.

Config gdiff_tool= COMMAND
    default gvimdiff
        Launch graphical diff tool.

Config geditor= COMMAND
    default gedit
        Launch graphical text editor.

Config image_viewer= COMMAND
    default eog
        Launch image viewer.

Config rsync= COMMAND
    default rsync -a --exclude='.*' --timeout=1800 -rsh='ssh -oBatchMode=yes'
        The rsync command.

Config ssh= COMMAND
    default ssh -oBatchMode=yes -oConnectTimeout=10
        The ssh command.

Config terminal= COMMAND
    default xterm
        Launch terminal.

Config [rosa-ldap]
    These LDAP settings are only relevant if \[rosa-svn\] user-tool=ldap (page 200).

Config attrs= uid cn mail
    default uid cn mail
        The attributes for UID, common name and email in the LDAP directory.

Config basedn= BASEDN
    The base DN in the LDAP directory to search for users.

Config binddn= BINDDN
    The DN in the LDAP server to bind with to search the directory.

Config filter-more
    If specified, use the value as extra (AND) filters to an LDAP search.

Config password-file
    default ~/.ldappw
        The file containing the password to the LDAP server, if required.

Config tls-ca-file

```

The location of the TLS CA certificate file.

**Config uri= URL**

The URI of the LDAP server e.g. ldap://my-ldap.my-domain.

**Config [rosa-svn]**

Configuration for rosa svn-pre-commit and rosa svn-post-commit.

**Config notification-from= EMAIL-ADDRESS**

Notification email address (for the “From.” field in notification emails).

**Config notify-who-on-trunk-commit= ROLE1 ...**

List of user roles to notify on trunk commits. A user role can be:

- author for the author of the change.
- owner for the owner of the affected suite.
- access-list for the users in the access-list of the affected suite.

---

**Note:** If the author is not in the list, (s)he will not be notified regardless of his/her role as suite owner or author of the access-list.

---

E.g.:

`notify-who-on-trunk-commit=author owner access-list`

**Config super-users= USER1 ...**

Space-separated list of admin users allowed to write to the trunk of any suite.

**Config user-tool= ldap|passwd**

Tool to obtain user information. Either LDAP or Unix getpwnam.

**Config [rose-ana]**

Configuration related to the built-in *rose\_ana* (page 286) application.

**Config grepper-report-limit= 42**

Limits the number of lines printed when using the rose.apps.ana\_builtin.grepper analysis class.

**Config kgo-database= .true.**

**default** .false.

Turns on the *Rose Ana Comparison Database* (page 152).

**Config method-path= /path/1 /path2**

Items to prepend to the search path for user methods.

**Config skip-if-all-files-missing= .true.**

Causes the rose.apps.ana\_builtin.grepper class to pass if all files to be compared are missing.

**Config [rose-bush]**

Configuration for Rose Bush server.

**Config cycles-per-page= NUMBER**  
**default** 100

Cycles list view: default number of cycles per page.

**Config host= NAME**

**default** The server’s host name.

An alternative host name.

**Config jobs-per-page= NUMBER**  
**default** 15

Job list view: default number of jobs per page.

**Config jobs-per-page-max= NUMBER**

**default** 300

Job list view: maximum number of jobs per page.

**Config logo= HTML-IMG-ATTRIBUTE ...**

Image logo attributes, can be any HTML <img/> tag attributes  
e.g: logo=src="http://server/my-rose-bush-logo.png"  
alt="My Rose Bush Logo".

**Config suites-per-page= NUMBER**

**default** 100

Suites list view: default number of suites per page.

**Config title= TITLE**

**default** "Rose Bush"

An alternative service title.

**Config view-size-max= BYTES**

**default** 10485760

File view: maximum viewable file size in bytes.

**Config [rose-config-diff]**

Configuration specific to *rose config-diff* (page 248).

**Config ignore{foo}= REGEX**

Shorthand expressions for ignore-setting regular expressions e.g. ^baz,  
bar,w[ib]ble.

**Config properties= title,ns,description,help**

**default** "title,ns,description,help"

Metadata properties to use.

**Config [rose-config-edit]**

Configuration specific to *rose config-edit* (page 250).

**Config icon-path-scheduler= PATH**

**default** /opt/cylc/images/icon.svg

Path to an image containing the suite engine icon. See *rose.config\_editor* for detail.

**Config project-url= URL**

**default** https://github.com/metomi/rose/

Hyperlink to the Rose project page.

**Config [rose-host-select]**

Configuration related to *rose host-select* (page 253).

**Config default= GROUP/HOST ...**

The default arguments to use for this command e.g. default=hpc.

**Config group{NAME}= GROUP/HOST ...**

Declare a named group of hosts e.g:

```
group{rose-vm}=rose-vm0 rose-vm1 rose-vm2 rose-vm3
group{hpc}=hpc1 hpc2
group{hpc1}=hpc1a hpc1b hpc1c hpc1d
group{hpc2}=hpc2a hpc2b hpc2c hpc2d
```

**Config method{NAME}= METHOD [ :METHOD-ARG ]**

**default** load

Declare the default ranking method for a group of hosts e.g.  
method{hpc}=random. For detail on the methods see *rose host-select* (page 253).

**Config thresholds{NAME}= [METHOD [:METHOD-ARG] :] VALUE ...**

Declare the default threshold(s) for a group of hosts e.g:

```
thresholds{hpc}=fs:/var/tmp:75 fs:/tmp:75  
thresholds{linux}=mem:8000
```

For detail on the methods see *rose host-select* (page 253).

**Config timeout= FLOAT**  
**default** 10.0

Set the timeout in seconds of ssh commands to hosts.

**Config [rose-mpi-launch]**

Configuration related to *rose mpi-launch* (page 257).

**Config launcher-fileopts.LAUNCHER= OPTION-TEMPLATE**

Specify the options to a LAUNCHER for launching with a command file. The template string should contain \$ROSE\_COMMAND\_FILE (or \${ROSE\_COMMAND\_FILE}), which will be expanded to the path to the command file. e.g:

```
launcher-fileopts.poe=-f $ROSE_COMMAND_FILE  
launcher-fileopts.mpiexec=-cmdfile $ROSE_COMMAND_FILE
```

**Config launcher-list= LAUNCHER ...**

Specify a list of launcher commands e.g:

```
launcher-list=poe mpiexec
```

**Config launcher-postopts.LAUNCHER= OPTIONS-TEMPLATE**

Specify the options to a LAUNCHER for launching with a command. postopts are options placed after COMMAND but before the remaining arguments.

**Config launcher-preopts.LAUNCHER= OPTIONS-TEMPLATE**

Specify the options to a LAUNCHER for launching with a command. preopts are options placed after the launcher command but before COMMAND. E.g:

```
launcher-preopts.mpiexec=-n $NPROC
```

**Config [rose-stem]**

Configuration related to *rose stem* (page 259).

**Config automatic-options= VARIABLE=VALUE**

Automatic options. These are added as if the user added them with --define-suite on the command line and can be accessed as Jinja2 variables in the suite.rc file. E.g automatic-options=TEA=earl\_grey would set the Jinja2 variable TEA to be earl\_grey.

**Config [rose-suite-log]**

Configuration related to *rose suite-log* (page 262).

**Config rose-bush= URL**

URL to the site's Rose Bush web service.

**Config [rose-suite-run]**

Configuration related to *rose suite-run* (page 263).

**Config hosts= HOST-GROUP|HOST ...**

Hosts in the *[rose-host-select]* (page 201) section that can be used to run a suite e.g:

```
hosts=rose-vm
```

**Config remote-no-login-shell= HOST-GLOB=true|false**  
**default false**

Don't use login shell to invoke rose suite-run --remote where HOST-GLOB is a glob for matching host names e.g:

```
remote-no-login-shell=myhpc*=true  

mycluster*=true
```

**Config remote-rose-bin= HOST-GLOB=ROSE-BIN-PATH**  
**default rose**

Path to rose executable on remote hosts where:

- HOST-GLOB is a glob for matching host names.
- ROSE-BIN-PATH is the path to the rose executable.

E.g:

```
remote-rose-bin=myhpc*/=/opt/rose/bin/rose  

mycluster*/=/usr/local/bin/rose
```

**Config root-dir= HOST-GLOB=\$HOST-DIR**  
**default \$HOME**

Root location of a suite run directory where:

- HOST-GLOB is a glob for matching host names.
- HOST-DIR is the value of the root location for matching hosts.

E.g:

```
root-dir=hpc*/=$DATADIR  

=*$HOME
```

**Config root-dir{share/cycle}= HOST-GLOB=\$HOST-DIR**  
**default \$HOME**

Root location of a suite run's share/cycle/ directory. Syntax is the same as *root-dir* (page 203). Multiple pairs can be specified by a new-line separated list.

**Config root-dir{share}= HOST-GLOB=\$HOST-DIR**  
**default \$HOME**

Root location of a suite run's share/ directory. Syntax is the same as *root-dir* (page 203). Multiple pairs can be specified by a new-line separated list.

**Config root-dir{work}= HOST-GLOB=\$HOST-DIR**  
**default \$HOME**

Root location of a suite run's work/ directory. Syntax is the same as *root-dir* (page 203). Multiple pairs can be specified by a new-line separated list.

**Config scan-hosts= HOST-GROUP | HOST ...**

Hosts in the *[rose-host-select]* (page 201) section that can be scanned by rose utilities e.g:

```
scan-hosts=localhost rose-vm
```

**Config [rose-task-run]**

Configuration related to *rose task-run* (page 267).

**Config path-prepend= /path/1 /path/2**

Items to prepend to the PATH environment variable.

**Config path-prepend.PYTHONPATH**

Items to prepend to a PATH-like environment variable, e.g. PYTHONPATH.

**Config [rosie-db]**

Configuration related to the database of the Rosie web service server.

**Config db.PREFIX= URL**

The database location of a given repository prefix, from within the server  
e.g. to create/reference an SQLite DB at /srv/rose/foodb.  
sqlite:

```
db.foo=sqlite:///srv/rose/foodb.sqlite
```

**Config repos.PREFIX= DIR**

The path to the repository of a given prefix, from within the server e.g:

```
repos.foo=/srv/svn/foo
```

**Config [rosie-disco]**

Configuration related to the adhoc Rosie suite discovery service server.

**Config host= NAME**

**default** The server's host name.

An alternative host name.

**Config title= TITLE**

**default** "Rosie Suites Discovery"

An alternative service title.

**Config [rosie-go]**

Configuration related to the *rosie go* (page 270) GUI. See `rosie.browser` for detail.

**Config icon-path-scheduler**

**default** /opt/cylc/images/icon.svg

Path to an image containing the suite engine icon.

**Config project-url= https://host/rose/**

**default** <https://github.com/metomi/rose/>

Hyperlink to the Rose project page.

**Config [rosie-id]**

Configuration related to Rosie client commands.

**Config local-copy-root= DIR**

**default** \$HOME/roses

Root directory of local (working) copies of suites e.g:

```
local-copy-root=$HOME/my-work/roses
```

**Config prefix-default= PREFIX**

The default ID prefix (an identifier for a Rosie repository/service).

**Config prefix-https-ssl-cert.PREFIX= PATH**

For HTTPS requests, location(s) of certificate + private key files e.g:

```
prefix-https-ssl-cert.PREFIX=/path/to/server.pem
prefix-https-ssl-cert.PREFIX=/path/to/server.crt /path/
→to/key
```

**Config prefix-https-ssl-verify.PREFIX= True|False**

**default** True

For HTTPS requests, verify SSL certificates.

**Config prefix-location.PREFIX= URL**

URL of the repository of an ID prefix, with no trailing slash, e.g:

```
prefix-location.foo=svn://host/foo
```

**Config prefix-password-store.PREFIX= gpgagent|gnomekeyring**  
**default** try gpgagent then gnomekeyring

The password store to use.

**Config prefix-username.PREFIX= USER-ID**

Default user name for services in prefix.

**Config prefix-web.PREFIX= URL**

Source browser (e.g. Trac) URL of the repository of an ID prefix e.g:

```
prefix-web.foo=http://host/projects/foo/intertrac/  
→source:
```

**Config prefix-ws.PREFIX= URL**

Discovery service URL of an ID prefix e.g:

```
prefix-ws.foo=http://host:port/rosie/foo
```

**Config prefixes-ws-default= PREFIX ...**

List of default discovery services (space-delimited list of prefixes) that will be used by a Rosie discovery service client.

**Config [rosie-vc]**

Configuration related to the Rosie version control client.

**Config access-list-default= USER-ID ...**

Default access-list setting on suite creation, e.g. to grant access to all users by default:

```
access-list-default=*
```

**Config [t]**

Test battery configuration.

**Config difftool= COMMAND ...**  
**default** diff -u

Tool to compare two files when there are differences.

**Config host-groups= GROUP ...**

List of selectable host groups.

**Config job-host= HOST**

A remote host that does not share its HOME directory with localhost.

**Config job-host-fast-dest-root= PATH**

A fast file system for working with small files in *job-host* (page 205) e.g:

```
job-host-fast-dest-root=/var/tmp/$USER
```

**Config job-host-with-share= HOST**

A remote host that shares its HOME directory with localhost.

**Config job-host-with-share-fast-dest-root= PATH**

A fast file system for working with small files in *job-host-with-share* (page 205).

**Config job-hosts= HOST1 HOST2**

Two non-local job hosts that may or may not share a file system.

```
Config job-hosts-sharing-fs= HOST1 HOST2
    Two job hosts sharing a file system, but not with localhost.
```

```
Config prove-options= PROVE-OPTIONS ...
    default -j9 -s
```

Options for `prove` when calling `rose test-battery` (page 267) with no arguments. Note that `-r t` is automatically added to the end of the command line.

## 7.4 Configuration Metadata

Configuration metadata uses the standard *Rose Configuration Format* (page 223). It is represented in a directory with the following:

- `rose-meta.conf` (page 207), the main metadata configuration file.
- `opt/` directory (see *Optional Configuration* (page 225)).
- Other files, e.g.:
  - `lib/python/widget/my_widget.py` would be the location of a *custom widget* (page 170).
  - `lib/python/macros/my_macro_validator.py` would be the location of a *custom macro* (page 126).
  - `etc/` would contain any resources for the logics in `lib/python/`, such as an icon for the custom widget.

Rose utilities will search for metadata using the following in order of precedence:

1. Configuration metadata embedded in the `meta/` directory of a suite or an application.
2. The `--meta-path=PATH` option of relevant commands.
3. The value of the `ROSE_META_PATH` (page 277) environment variable.
4. The `rose.conf/meta-path` (page 198) setting (see *Site And User Configuration* (page 198)).

---

**Tip:** See *Appendix: Metadata Location* (page 216) for more details.

---

The configuration metadata that controls default behaviour will be located in `$ROSE_HOME/etc/rose-meta/`.

### 7.4.1 Configuration Metadata File

The `rose-meta.conf` (page 207) contains a serialised data structure that is an unordered map (`sections=`) of unordered maps (`keys=values`).

The section name in a configuration metadata file should be a unique ID to the relevant configuration setting. The syntax of the ID is `SECTION-NAME=OPTION-NAME` or just `SECTION-NAME`. For example, `env=MY_ENV_NAME` is the ID of an environment variable called `MY_ENV_NAME` in an application configuration file; `namelist:something_nl=variable_name1` is the ID of a namelist variable called `variable_name1` in a namelist group called `something_nl` in an application configuration file. If the configuration metadata applies to a section in a configuration file, the ID is just the section name.

Where multiple instances of a section are used in a configuration file, ending in brackets with numbers, the metadata ID should just be based on the original section name (for example `namelist:extract_control(2)` should be found in the metadata under `namelist:extract_control`).

Finally, the configuration metadata may group settings in namespaces, which may in turn have common configuration metadata settings. The ID for a namespace set in the metadata is `ns=NAME`, e.g. `ns=env/MyFavouriteEnvironmentVars`.

## Metadata Inheritance (Import)

A root level `import=MY_COMMAND1/VERSION1 MY_COMMAND2/VERSION2 ... setting` in the `rose-meta.conf` (page 207) file will tell Rose metadata utilities to locate the meta keys `MY_COMMAND1/VERSION1`, `MY_COMMAND2/VERSION2` (and so on) and inherit their configuration and files if found.

For example, you might have a `rose-meta` directory that contains the following files and directories:

```
cheese_sandwich/
  vn1.5/
    rose-meta.conf
  vn2.0/
    rose-meta.conf
cheese/
  vn1.0/
    rose-meta.conf
```

and write an app referencing this `rose-meta` directory that looks like this:

```
meta=cheese_sandwich/vn2.0

[env]
CHEESE=camembert
SANDWICH_BREAD=brown
```

This will reference the metadata at `rose-meta/cheese_sandwich/vn2.0`.

Now, we can write the `rose-meta.conf` (page 207) file using an import:

```
import=cheese/vn1.0

[env=SANDWICH_BREAD]
values=white,brown,seeded
```

which will inherit metadata from metadata from `rose-meta/cheese/vn1.0/rose-meta.conf`.

## Metadata Options

The metadata options for a configuration fall into four categories: `sorting` (page 207), `values` (page 208), `behaviour` (page 212) and `help` (page 215) as outlined below.

### Rose Configuration `rose-meta.conf`

#### **Config [SETTING]**

A section containing metadata items relating to a particular setting.

SETTING should be the full name of a configuration containing the name of the section and the name of the setting separated by an equals = sign e.g:

- `[env=FOO]` would refer to the environment variable FOO
- `[namelist:foo=BAR]` would refer to BAR from the namelist foo.

#### **Metadata for Sorting**

These configuration metadata are used for grouping and sorting the IDs of the configurations.

**Config ns**

A forward slash / delimited hierarchical namespace for the container of the setting, which overrides the default. The default namespace for the setting is derived from the first part of the ID - by splitting up the section name by colons : or forward slashes /. For example, a configuration with an ID namelist:var\_minimise=niter\_set would have the namespace namelist/ var\_minimise. If a namespace is defined for a section, it will become the default for all the settings in that section.

The namespace is used by *rose config-edit* (page 250) to group settings, so that they can be placed in different pages. A namespace for a section will become the default for all the settings in that section.

---

**Note:** You should not assign namespaces to variables in duplicate sections.

---

**Config sort-key**

A character string that can be used as a sort key for ordering an option within its namespace.

It can also be used to order sections and namespaces.

The *sort-key* (page 208) is used by *rose config-edit* (page 250) to group settings on a page. Items with a *sort-key* (page 208) will be sorted to the top of a name-space. Items without a *sort-key* (page 208) will be sorted after, in ascending order of their IDs.

The sorting procedure in pseudo code is a normal ASCII-like sorting of a list of setting\_sort\_key + "~" + setting\_id strings. If there is no setting\_sort\_key, null string will be used.

For example, the following metadata:

```
[env=apple]
[env=banana]
[env=cherry]
sort-key=favourites-01

[env=melon]
sort-key=do-not-like-01

[env=prune]
sort-key=do-not-like-00
```

would produce a sorting order of env=prune, env=melon, env=cherry, env=apple, env=banana.

## Metadata for Values

These configuration metadata are used to define the valid values of a setting. A Rose utility such as *rose config-edit* (page 250) can use these metadata to display the correct widget for a setting and to check its input. However, if the value of a setting contains a string that looks like an environment variable, these metadata will normally be ignored.

**Config type****Default raw**

The type/class of the setting. The type names are based on the intrinsic Fortran types, such as `integer` and `real`. Currently supported types are:

**boolean** *example option:* `PRODUCE_THINGS=true`

*description:* either `true` or `false`

*usage:* environment variables, javascript/JSON inputs

**character** *example option:* sea\_colour='blue'  
*description:* Fortran character type - a single quoted string, single quotes escaped in pairs  
*usage:* Fortran character types

**integer** *example option:* num\_lucky=5  
*description:* generic integer type  
*usage:* any integer-type input

**logical** *example option:* l\_spock=.true.  
*description:* Fortran logical type - either .true. or .false.  
*usage:* Fortran logical types

**python\_boolean** *example option:* ENABLE\_THINGS=True  
*description:* Python boolean type - either True or False  
*usage:* Python boolean types

**python\_list** *description:* used to signify a Python-compatible formatted list such as ["Foo", 50, False].

**Warning:** This encapsulates length, so do not use a separate length declaration for this setting.

*usage:* use for inputs that expect a string that looks like a Python list - e.g. Jinja2 list input variables.

**quoted** *example option:* js\_measure\_cloud\_mode="laser"  
*description:* a double quoted string, double quotes escaped with backslash  
*usage:* Inputs that require double quotes and allow backslash escaping e.g. javascript/JSON inputs.

**real** *example option:* n\_avogadro=6.02e23  
*description:* Fortran real number type, generic floating point numbers  
*usage:* Fortran real types, generic floating point numbers.

---

**Note:** Scientific notation must use the “e” or “E” format.

---

*comment:* Internally implemented within Rose using Python’s floating point (<https://docs.python.org/3/library/stdtypes.html#types-numeric>) specification.

**raw** *description:* placeholder used in derived type specifications where none of the above types apply  
*usage:* only in derived types

**spaced\_list** *description:* used to signify a space separated list such as "Foo" 50 False.  
*usage:* use for inputs that expect a string that contains a number of space separated items - e.g. in fcm\_make app configs.

---

**Note:** Not all inputs need to have type defined. In some cases using values or pattern is better.

---

A derived type may be defined by a comma , separated list of intrinsic types, e.g. integer, character, real, integer. The default is a raw string.

#### Config length

Define the length of an array. If not present, the setting is assumed to be a scalar. A positive integer defines a fixed length array. A colon : defines a dynamic length array.

---

**Note:** You do not need to use `length` (page 209) if you already have `type=python_list` (page 208) for a setting.

---

#### **Config element-titles**

Define a list of comma separated “titles” to appear above array entries. If not present then no titles are displayed.

---

**Note:** Where the number of `element-titles` (page 210) is greater than the length of the array, it will only display titles up to the length of the array. Additionally, where the associated array is longer than the number of `element-titles` (page 210), blank headings will be placed above them.

---

#### **Config values**

Define a comma , separated list of permitted values of a setting (or an element in the setting if it is an array). This metadata overrides the `type` (page 208), `range` (page 210) and `:rose:conf:pattern` metadata.

For example, `rose config-edit` (page 250) may use this list to determine the widget to display the setting. It may display the choices using a set of radio buttons if the list of values is small, or a drop down combo box if the list of `values` (page 210) is large. If the list only contains one value, `rose config-edit` (page 250) will expect the setting to always have this value, and may display it as a special setting.

#### **Config value-titles**

Define a comma , separated list of titles to associate with each of the elements of `values` (page 210) which will be displayed instead of the value. This list should contain the same number of elements as the `values` (page 210) entry.

For example, given the following metadata:

```
[env=HEAT]
values=0, 1, 2
value-titles=low, medium, high
```

`rose config-edit` (page 250) will display low for option value 0, medium for 1 and high for 2.

#### **Config value-hints**

Define a comma , separated list of suggested values for a variable as value “hints”, but still allows the user to provide their own override. This is like an auto-complete widget.

For example, given the following metadata:

```
[env=suggested_fruit]
value-hints=pineapple, cherry, banana, apple, pear, mango, kiwi, grapes, peach,
    ↪fig,
    =orange, strawberry, blackberry, blackcurrent, raspberry, melon,
    ↪plum
```

`rose config-edit` (page 250) will display possible option values when the user starts typing if they match a suggested value.

#### **Config range**

Specify a range of values. It can either be a simple comma , separated list of allowed values, or a logical expression in the Rose metadata `mini-language` (page 219). This metadata is only valid if `type` (page 208) is either `integer` or `real`.

A simple list may contain a mixture of allowed numbers and number ranges such as `1, 2, 4:8, 10:` (which means the value can be 1, 2, between 4 and 8 inclusive, or any values greater than or equal to 10.)

A logical expression uses the Rose metadata *mini-language* (page 219), using the variable `this` to denote the value of the current setting, e.g. `this <1` and `this >1`.

**Warning:** Inter-variable comparisons are not permitted (but see the `fail-if` property below for a way to implement this).

### Config pattern

Specify a regular expression (Python Regular Expressions (<https://docs.python.org/2/library/re.html#regular-expression-syntax>)) to compare against the whole value of the setting.

For example, if we write the following metadata:

```
[namelist:cheese=country_of_origin]
pattern=^"[A-Z].+"$
```

then we expect all valid values for `country_of_origin` to start with a double quote (^"), begin with an uppercase letter ([A-Z]), contain some other characters or spaces (. +), and end with a quote ("\$).

If you have an array variable (for example, `TARTAN_PAINT_COLOURS='blue','red','blue'`) and you want to specify a pattern that each element of the array must match, you can construct a regular expression that repeats and includes commas. For example, if each element in our `TARTAN_PAINT_COLOURS` array must obey the regular expression '`red'|'blue'`', then we can write:

```
[env=TARTAN_PAINT_COLOURS]
length=:
pattern=^('red'|'blue')(?:, ('red'|'blue'))*$
```

### Config fail-if

Specify a logical expression using the Rose *mini-language* (page 219) to validate the value of the current setting with respect to other settings. If the logical expression evaluates to true, the system will consider the setting in error.

See the associated setting `warn-if` (page 212) for raising warnings.

The logical expression uses a Python-like syntax (documented in the [appendix](#) (page 219)). It can reference the value of the current setting with the `this` variable and the value of other settings with their IDs. E.g.:

```
[namelist:test=my_test_var]
fail-if=this < namelist:test=control_lt_var;
```

means that an error will be flagged if the numeric value of `my_test_var` is less than the numeric value of `control_lt_var`.

```
fail-if=this != 1 + namelist:test=ctrl_var_1 * (namelist:test=ctrl_var_
→2 - this);
```

shows a more complex operation, again with numeric values.

To check array elements, the ID should be expressed with a bracketed index, as in the configuration:

```
fail-if=this(2) != "'0A'" and this(4) == "'0A'";
```

---

**Note:** With array elements indexing starts from 1.

---

Array elements can also be checked using `any` and `all`. E.g.:

```
fail-if=any(namelist:test=ctrl_array < this);
fail-if=all(this == 0);
```

Similarly, the number of array elements can be checked using `len`. E.g.:

```
fail-if=len(namelist:test=ctrl_array) < this;
fail-if=len(this) == 0;
```

Expressions can be chained together and brackets used:

```
fail-if=this(4) == "'0A'" and (namelist:test=ctrl_var_1 != "'N'" or
namelist:test=ctrl_var_2 != "'Y'" or all(namelist:test=ctrl_arr_3 == 'N
↳'));
```

Multiple failure conditions can be added, by using a semicolon as the separator - the semicolon is optional for a single statement or the last in a block, but is recommended. Multiple failure conditions are essentially similar to chaining them together with `or`, but the system can process each expression one by one to target the error message.

```
fail-if=this > 0; this % 2 == 1; this * 3 > 100;
```

You can add a message to the error or warning message to make it clearer by adding a hash followed by the comment at the end of a configuration metadata line:

```
# Need common divisor for ctrl_array
fail-if=any(namelist:test=ctrl_array % this == 0);
```

When using multiple failure conditions, different messages can be added if they are placed on individual lines:

```
fail-if=this > 0; # Needs to be less than or equal to 0
                  this % 2 == 1; # Needs to be odd
                  this * 3 > 100; # Needs to be more than 100/3.
```

---

**Note:** When dividing a real-numbered setting by something, make sure that the expression does not actually divide an integer by an integer - for example, `this / 2` will evaluate as 0 if `this` has a value of 1, but 0.5 if it has a value of 1.0. This is a result of Python's implicit typing.

You can get around this by making sure either the numerator or denominator is a real number - e.g. by rewriting it as `this / 2.0` or `1.0 * this / 2`.

---

#### Config `warn-if`

Specify a logical expression using the Rose *mini-language* (page 219) to validate the value of the current setting with respect to other settings. If the logical expression evaluates to true, the system will issue a warning. It is a slightly different usage of the `fail-if` functionality which can do things like warn of deprecated content, e.g.:

```
warn-if=True;
```

would always evaluate `True` and give a warning if the setting is present.

See the associated setting `fail-if` (page 211) for examples of logical expressions that may be added.

#### Metadata for Behaviour

These metadata are used to define how the setting should behave in different states.

### **Config compulsory**

A true value indicates that the setting should be compulsory. If this flag is not set, the setting is optional.

Compulsory sections should be physically present in the configuration at all times. Compulsory options should be physically present in the configuration if their parent section is physically present.

Optional settings can be removed as the user wishes. Compulsory settings may however be triggered ignored (see [trigger](#) (page 213)). For example, the [rose config-edit](#) (page 250) may issue a warning if a compulsory setting is not defined. It may also hide a compulsory variable that only has a single permitted value.

### **Config trigger**

A setting has the following states:

- normal
- user ignored (stored in the configuration file with a ! flag, ignored at run time)
- logically ignored (stored in the configuration file with a !! flag, ignored at runtime)

If a setting is user ignored, the trigger will do nothing. Otherwise:

- If the logic for a setting in the trigger is fulfilled, it will cause the setting to be enabled.
- If it is not, it will cause the setting to be logically ignored.

The trigger expression is a list of settings triggered by (different values of) this setting. If the values are not given, the setting will be triggered only if the current setting is enabled.

The syntax contains ID-value pairs, where the values part is optional. Each pair must be separated by a semi-colon. Within each pair, any values must be separated from the ID by a colon (:) and a space. Values must be formatted in the same way as the setting [values](#) (page 210) defined above (i.e. comma separated).

The trigger syntax looks like:

```
[namelist:trig_nl=trigger_variable]
trigger=namelist:dep_nl=A;
namelist:dep_nl=B;
namelist:value_nl=X: 10;
env=Y: 20, 30, 40;
namelist:value_nl=Z: 20;
```

In this example:

- When namelist:trig\_nl=trigger\_variable is ignored, all the variables in the trigger expression will be ignored, irrespective of its value.
- When namelist:trig\_nl=trigger\_variable is enabled, namelist:dep\_nl=A and namelist:dep\_nl=B will both be enabled, and the other variables may be enabled according to its value:
  - When the value of the setting is not 10, 20, 30, or 40, namelist:value\_nl=X, env=Y and namelist:value\_nl=Z will be ignored.
  - When the value of the setting is 10, namelist:value\_nl=X will be enabled, but env=Y and namelist:value\_nl=Z will be ignored.
  - When the value of the setting is 20, env=Y and namelist:value\_nl=Z will be enabled, but namelist:value\_nl=X will be ignored.
  - When the value of the setting is 30, env=Y will be enabled, but namelist:value\_nl=X and namelist:value\_nl=Z will be ignored.
  - If the value of the setting contains an environment variable-like string, e.g. \${TEN\_MULTIPLE}, all three will be enabled.

Settings mentioned in trigger expressions will have their default state set to ignored unless explicitly triggered. [rose config-edit](#) (page 250) will issue warnings if variables or sections are in the incorrect state when it loads a configuration. Triggering thereafter will work as normal.

Where multiple triggers act on a setting, the setting is triggered only if all triggers are active (i.e. an AND relationship). For example, for the two triggers here:

```
[env=IS_WATER]
trigger=env=IS_ICE: true;

[env=IS_COLD]
trigger=env=IS_ICE: true;
```

the setting `env=IS_ICE` is only enabled if both `env=IS_WATER` and `env=IS_COLD` are true and enabled. Otherwise, it is ignored.

The trigger syntax also supports a logical expression using the Rose metadata *mini-language* (page 219), in the same way as the `range` (page 210) or `fail-if` (page 211) metadata. As with `range` (page 210), inter-variable comparisons are disallowed.

```
[env=SNOWFLAKE_SIDES]
trigger=env=CUSTOM_SNOWFLAKE_GEOMETRY: this != 6;
env=SILLY_SNOWFLAKE_GEOMETRY: this < 2;
```

In this example, the setting `env=CUSTOM_SNOWFLAKE_GEOMETRY` is enabled if `env=SNOWFLAKE_SIDES` is enabled and not 6. `env=SILLY_SNOWFLAKE_GEOMETRY` is only enabled when `env=SNOWFLAKE_SIDES` is enabled and less than 2. The logical expression syntax can be used with non-numeric variables in the same way as the fail-if metadata.

#### **Config duplicate**

Allow duplicated copies of the setting. This is used for sections where there may be more than one with the same metadata - for example multiple namelist groups of the same name. If this setting is true for a given name, the application configuration will accept multiple namelist groups of this name. `rose config-edit` (page 250) may then provide the option to clone or copy a namelist to generate an additional namelist. Otherwise, `rose config-edit` (page 250) may issue warning for configuration sections that are found with multiple copies or an index.

#### **Config macro**

Associate a setting with a comma-delimited set of custom macros (but not upgrade macros).

E.g. for a macro class called `FibonacciChecker` in the metadata under `lib/python/macros/fib.py`, we may have:

```
macro=fib.FibonacciChecker
```

This may be used in `rose config-edit` (page 250) to visually associate the setting with these macros. If a macro class has both a `transform` and a `validate` method, you can specify which you need by appending the method to the name e.g.:

```
macro=fib.Fibonacci.validate
```

#### **Config widget [gui-application]**

Indicate that the gui-application (e.g. `rose config-edit` (page 250)) should use a special widget to display this setting.

E.g. If we want to use a slider instead of an entry box for a floating point real number.

The widget may take space-delimited arguments which would be specified after the widget name. E.g. to set up a hypothetical table with named columns X, Y, VCT, and Level, we may do:

```
widget[rose-config-edit]=table.TableWidget X Y VCT Level
```

You may override to a Rose built-in widget by specifying a full `rose` class path in Python - for example, to always show radiobuttons for an option with `values` (page 210) set:

```
widget[rose-config-edit]=rose.config_editor.valuewidget.radiobuttons.
→RadioButtonsValueWidget
```

Another useful Rose built-in widget to use is the array element aligning [page widget](#) (page 297), `rose.config_editor.pagewidget.table`. You can set this for a section or namespace to make sure each  $n$ -th variable value element lines up horizontally. For example:

```
[namelist:meal_choices]
customers='Athos','Porthos','Aramis','d''Artagnan'
entrees='soup','pate','soup','asparagus'
main='beef','spaghetti','coq au vin','lamb'
petits_fours=.false.,.true.,.false.,.true.
```

could use the following metadata:

```
[namelist:meal_choices]
widget[rose-config-edit]=rose.config_editor.pagewidget.table.
→PageArrayTable
```

to align the elements on the page like this:

customers	Athos	Porthos	Aramis	d'Artagnan
entrees	soup	pate	soup	asparagus
main	beef	spaghetti	coq au vin	lamb
petits_fours	.false.	.true.	.false.	.true.

#### **Config copy-mode**

For use with settings in the [rose-suite.info](#) (page 197) file.

Setting [copy-mode](#) (page 215) in the metadata allows for the field to be either never copied or copied with any value associated to be `clear`.

For example: in a [rose-suite.info](#) (page 197) file:

```
[ensemble members]
copy-mode=never
```

Setting the ensemble members field to include `copy-mode=never` means that the ensemble members field would never be copied.

```
[additional info]
copy-mode=clear
```

Setting the additional info field to include `copy-mode=never` means that the additional info field would be copied, but any value associated with it would be cleared.

## **Metadata for Help**

These metadata provide help for a configuration.

#### **Config url**

A web URL containing help for the setting. For example:

```
url=http://www.something.com/FOO/view/dev/doc/FOO.html
```

For example, the [rose config-edit](#) (page 250) will trigger a web browser to display this when a variable name is clicked. A partial URL can be used for variables if the variable's section or namespace has a relevant parent `url` property to use as a prefix. For example:

```
[namelist:foo]
url=https://www.google.com/search

[namelist:foo=bar]
url=?q=nearest+bar
```

**Config help**

(Long) help for the setting. For example, *rose config-edit* (page 250) will use this in a pop-up dialog for a variable. Embedding variable IDs in the help string will allow links to the variables to be created within the pop-up dialog box, e.g.

```
help=Used in conjunction with namelist:Var_DiagnosticsNL=n_linear_adj_
→test to do something linear.
```

Web URLs beginning with `http://` will also be presented as links in the *rose config-edit* (page 250).

**Config description**

(Medium) description for the setting. For example, *rose config-edit* (page 250) will use this as part of the hover over text.

*rose config-edit* (page 250) will also use descriptions set for sections or namespaces as page header text (appears at the top of a panel or page), with clickable ID and URL links as in help. Descriptions set for variables may be automatically shown underneath the variable name in *rose config-edit* (page 250), depending on view options.

**Config title**

(Short) title for the setting. For example, *rose config-edit* (page 250) can use this specification as the label of a setting, instead of the variable name.

## 7.4.2 Appendix: Metadata Location

Centralised Rose metadata is referred to with either the *rose-suite.conf/meta* (page 197) or *rose-suite.info/project* (page 198) settings in a suite configuration. It needs to live in a system-global-readable location.

Rose utilities will do a path search for metadata using the following in order of precedence:

- The `--meta-path=PATH` option of relevant commands.
- The content of the *ROSE\_META\_PATH* (page 277) environment variable.
- The *rose.conf/meta-path* (page 198) setting (see *Site And User Configuration* (page 198)).

Each of the above settings can be a colon-separated list of paths.

Underneath each directory in the search path should be a hierarchy like the following:

```
 ${APP} /HEAD/
 ${APP} /${VERSION} /
 ${APP} /versions.py # i.e. the upgrade macros
```

---

**Note:** A *rose-suite.info* (page 197) is likely to have no versions.

---

---

**Note:** In some cases, a number of different executables may share the same application configuration metadata in which case APP is given a name which covers all the uses.

---

---

**Tip:** The Rose team recommend placing metadata in a *rose-meta* directory at the top of a project's source tree. Central metadata, if any, at the *meta-path* location in the site configuration, should be a collection of regularly-updated subdirectories from all of the relevant projects' *rose-meta* directories.

For example, a system CHOCOLATE may have a flat metadata structure within the repository:

```
CHOCOLATE/doc/
...
CHOCOLATE/rose-meta/
CHOCOLATE/rose-meta/choc-dark/
CHOCOLATE/rose-meta/choc-milk/
```

and the system CAFFEINE may have a hybrid structure, with both flat and hierarchical components:

```
CAFFEINE/doc/
...
CAFFEINE/rose-meta/caffeine-guarana/
CAFFEINE/rose-meta/caffeine-coffee/cappuccino/
CAFFEINE/rose-meta/caffeine-coffee/latte/
CAFFEINE/rose-meta/caffeine-tea/yorkshire/
CAFFEINE/rose-meta/caffeine-tea/lapsang/
```

and a site configuration with:

```
meta-path=/path/to/rose-meta
```

We would expect the following directories in /path/to/rose-meta:

```
/path/to/rose-meta/caffeine-guarana/
/path/to/rose-meta/caffeine-coffee/
/path/to/rose-meta/caffeine-tea/
/path/to/rose-meta/choc-dark/
/path/to/rose-meta/choc-milk/
```

with caffeine-coffee containing subdirectories cappuccino and latte, and caffeine-tea containing yorkshire and lapsang.

### 7.4.3 Upgrade and Versions

Terminology:

**The HEAD (i.e. development) version** The configuration metadata most relevant to the latest revision of the source tree.

**A named version** The configuration metadata most relevant to a release, or a particular revision, of the software. This will normally be a copy of the HEAD version at a given revision, although it may be updated with some backward compatible fixes.

Each change in the HEAD version that requires an upgrade procedure should introduce an upgrade macro. Each upgrade macro will provide the following information:

- A tag of the configuration which can be applied by this macro (i.e. the previous tag).
- A tag of the configuration after the transformation.

This allows our system to build up a chain if multiple upgrades need to be applied. The tag can be any name, but will normally refer to the ticket number that introduces the change.

Every new upgrade macro creates a new tagged version. A named version is simply a tagged version for which a copy of the relevant configuration metadata is made available.

Named versions for system releases are typically created at the end of the release process. The associated upgrade macro is typically only required in order to create the new name tag and, therefore, does not normally alter the application configuration.

Application configurations can reference the configuration metadata as follows:

```
#!cfg
# Refer to the HEAD version
# (typically you wouldn't do this since no upgrade process is possible)
# For flat metadata
meta=my-command
# For hierarchical metadata
meta=/path/to/metadata/my-command/HEAD

# Refer to a named or tagged version in the flat metadata
meta=my-command/8.3
meta=my-command/t123
# Refer to a named or tagged version in the hierarchical metadata
meta=/path/to/metadata/my-command/8.3
```

If a version is defined then the Rose utilities will first look for the corresponding named version. If this cannot be found then the HEAD version is used and, if an upgrade is available, a warning will be given to indicate that the configuration metadata being used requires upgrade macros to be run. If the version defined does not correspond to a tagged version then a warning will be given.

---

**Note:** If a hierarchical structure for the metadata is being used, the HEAD tag must be specified explicitly.

---

### When to create named versions

One option is to create a new named version for each release of your system. This makes it easy for users to understand. However, if there is a new release which does not require a change to the metadata then you will still have to create a new copy and force the user to go through a null upgrade which may not be desirable. An alternative is to only create a new named version at releases which require changes. The name then indicates the metadata is relevant for a particular release and all subsequent releases (unless an upgrade macro is available to a later release).

---

**Tip:** It is also possible to make any tagged version between releases a named version, but it will usually be better not to. In which case, the user will be using HEAD and will be prompted to upgrade (which is probably what you want if you're not using a release).

---

### Sharing metadata between different executables

If two different commands share the majority of their inputs then you may choose to use the same configuration metadata for both commands. Any differences (in terms of available inputs) can then be triggered by the command in use. Whether this is desirable will partly depend on how many of the inputs are shared.

One downside of sharing metadata is that your application configuration may contain (ignored) settings which have no relevance to the command you are using.

---

**Note:** We intend to introduce support for configuration metadata to include / inherit from other metadata. This may mean that it makes sense to have separate metadata for different commands even when the majority of inputs are shared.

---

Another reason you may want to share metadata is if you have two related commands which you want to configure using the same set of inputs (i.e. a single application configuration).

This works by setting an alternate command in the application configuration and then using the --command-key option to [rose app-run](#) (page 245).

## Using development versions of upgrade macros

Users will be able to test out development versions of the upgrade macros by adding a working copy of the relevant branch into their metadata search path. However, care must be taken when doing this. Running the upgrade macro will change the `rose-app.conf/meta` (page 194) setting to refer to the new tag. If the upgrade macro is subsequently changed or other upgrade macros are added to the chain prior to this tag (because they get committed to the trunk first) then this will result in application configurations which have not gone through the correct upgrade process. Therefore, when using development versions of the upgrade macros it is safest to not commit any resulting changes (or to use a branch of the suite which you are happy to discard).

### 7.4.4 Metadata Mini-Language

The Rose metadata mini-language supports writing a logical expression in Python-like syntax, using variable IDs to reference their associated values.

Expressions are set as the value of metadata properties such as `rose-meta.conf[SETTING]fail-if` (page 211) and `rose-meta.conf[SETTING]range` (page 210).

The language is a small sub-set of Python - a limited set of operators is supported.

**Warning:** No built-in object methods, functions, or modules are supported - neither are control blocks such as `if/for`, statements such as `del` or `with`, or defining your own functions or classes. Anything that requires that kind of power should be in proper Python code as a macro.

Nevertheless, the language allows considerable power in terms of defining simple rules for variable values.

#### Operators

The following *numeric* operators are supported:

```
+      # add
-      # subtract
*      # multiply
**     # power or exponent - e.g. 2 ** 3 implies 8
/      # divide
//     # integer divide (floor) - e.g. 3 // 2 implies 1
%      # remainder e.g. 5 % 3 implies 2
```

The following *string* operators are supported:

```
+      # concatenate - e.g. "foo" + "bar" implies "foobar"
*      # self-concatenate some number of times - e.g. "foo" * 2 implies "foofoo"
%      # formatting - e.g. "foo %s baz" % "bar" implies "foo bar baz"
in     # contained in (True/False) - e.g. "oo" in "foobar" implies True
not in # opposite sense of in

# Where m, n are integers or expressions that evaluate to integers
# (negative numbers count from the end of the string):
[n]    # get nth character from string - e.g. "foo"[1] implies "o"
[m:n]  # get slice of string from m to n - e.g. "foobar"[1:5] implies "ooba"
[m:]   # get slice of string from m onwards - e.g. "foobar"[1:] implies "obar"
[:n]   # get slice of string up to n - e.g. "foobar":[5] implies "fooba"
```

The following *logical* operators are supported:

```
and   # Logical AND
or    # Logical OR
not   # Logical NOT
```

The following *comparison* operators are supported:

```
is      # Is the same object as (usually used for 'is none')
<      # Less than
>      # Greater than
==     # Equal to
>=     # Greater than or equal to
<=     # Less than or equal to
!=     # Not equal to
```

Operator precedence is intended to be the same as Python. However, with the current choice of language engine, the % and // operators may not obey this - make sure you force the correct behaviour using brackets.

## Constants

The following are special constants:

```
None  # Python None
False # Python False
True  # Python True
```

## Using Variable IDs

Putting a variable ID in the expression means that when the expression is evaluated, the string value of the variable is cast ([https://docs.python.org/2/library/ast.html#ast.literal\\_eval](https://docs.python.org/2/library/ast.html#ast.literal_eval)) and substituted into the expression.

For example, if we have a configuration that looks like this:

```
[namelist:zoo]
num_elephants=2
elephant_mood='peaceful'
```

and an expression in the configuration metadata:

```
namelist:zoo=elephant_mood != 'annoyed' and num_elephants >= 2
```

then the expression would become:

```
'peaceful' != 'annoyed' and 2 >= 2
```

If the variable is not currently available (ignored or missing) then the expression cannot be evaluated. If inter-variable comparisons are not allowed for the expression's parent option (such as with *rose-meta.conf[SETTING]trigger* (page 213) and *rose-meta.conf[SETTING]range* (page 210)) then referencing other variable IDs is not allowed.

In this case the expression would be false.

You may use *this* as a placeholder for the current variable ID - for example, the fail-if expression:

```
[namelist:foo=bar]
fail-if=namelist:foo=bar > 100 and namelist:foo=bar % 2 == 1
```

is the same as:

```
[namelist:foo=bar]
fail-if=this > 100 and this % 2 == 1
```

## Arrays

The syntax has some special ways of dealing with variable values that are arrays - i.e. comma-separated lists.

You can refer to a single element of the value for a given variable ID (or `this`) by suffixing a number in round brackets - e.g.:

```
namelist:foo=bar(2)
```

references the second element in the value for `bar` in the section `namelist:foo`. This follows Fortran index numbering and syntax, which starts at 1 rather than 0, i.e. `foo(1)` references the first element in the array `foo`.

If we had a configuration:

```
[namelist:foo]
bar='a', 'b', 'c', 'd'
```

`namelist:foo=bar(2)` would get substituted in an expression with '`b`' when the expression was evaluated. For example, an expression that contained:

```
namelist:foo=bar(2) == 'c'
```

would be evaluated as:

```
'b' == 'c'
```

Should you wish to make use of the array length in an expression you can make use of the `len` function, which behaves in the same manner as the Python `len` and Fortran `size` equivalents to return the array length. For example:

```
len(namelist:foo=bar) > 3
```

would be expanded to:

```
4 > 3
```

and evaluate as true.

There are two other special array functions, `any` and `all`, which behave in a similar fashion to their Python and Fortran equivalents, but have a different syntax.

They allow you to write a shorthand expression within an `any()` or `all()` bracket which implies a loop over each array element. For example:

```
any(namelist:foo=bar == 'e')
```

evaluates true if *any* elements in the value of `bar` in the section `namelist:foo` are '`e`'. For the above configuration snippet, this would be expanded before evaluation to be:

```
'a' == 'e' or 'b' == 'e' or 'c' == 'e' or 'd' == 'e'
```

Similarly,

```
all(namelist:foo=bar == 'e')
```

evaluates true if *all* elements are '`e`'. Again, with the above configuration, this would be expanded before proper evaluation:

```
'a' == 'e' and 'b' == 'e' and 'c' == 'e' and 'd' == 'e'
```

## Internals

Rose uses an external engine to evaluate the raw language string after variable IDs and any `any()` and `all()` functions have been substituted and expanded.

The current choice of engine is [Jinja2](http://jinja.pocoo.org/) (<http://jinja.pocoo.org/>), which is responsible for the details of the supported Pythonic syntax. This may change.

**Warning:** Do not use any Jinja2-specific syntax.

### 7.4.5 Config Editor Ignored Mechanics

This describes the intended behaviour of `rose config-edit` (page 250) when there is an ignored state mismatch for a setting - e.g. a setting might be enabled when it should be trigger-ignored.

`rose config-edit` (page 250) differs from the strict command line macro equivalent (`rose macro` (page 254)) because the *Switch Off Metadata* mode and accidentally metadata-less configurations need to be presented in a nice way without lots of unnecessary errors. `rose config-edit` (page 250) should only report the errors where the state is definitely wrong or makes a material difference to the user.

The table contains error and fixing information for some varieties of ignored state mismatches. The actual situations are considerably more varied, given section-ignoring and latent variables - the table holds the most important varieties.

The State contains the actual states. The Trigger State column contains the trigger-mechanism's expected states. The states can be:

**IT - !!** trigger ignored

**IU - !** user ignored

**E - (normal)** enabled

A subset of possible ignored/enabled states, errors and fixes:

State	Trigger State	Compulsory	Display Error?	User Options	Notes
IT	IT	compulsory	no	None	
IT	IT	optional	no	None	
IT	E	compulsory	error	E	
IT	E	optional	error	E	
IT	not trigger	compulsory	error	E	
IT	not trigger	optional	overlook	E	See <sup>1</sup>
IU	IT	compulsory	overlook	None	See <sup>2</sup>
IU	IT	optional	no	None	
IU	E	compulsory	error	E	
IU	E	optional	no	E	
IU	not trigger	compulsory	error	E	
IU	not trigger	optional	no	E	
E	IT	compulsory	error	IT	
E	IT	optional	error	IT	
E	E	compulsory	no	None	
E	E	optional	no	IU	
E	not trigger	compulsory	no	None	
E	not trigger	optional	no	IU	

<sup>1</sup> Overlooking mainly in order to de-clutter the No Metadata view.

<sup>2</sup> Same basic state - macro will ask to fix this.

## 7.5 Rose Configuration Format

A configuration in Rose is normally represented by a directory with the following:

- a configuration file in a modified [INI](https://en.wikipedia.org/wiki/INI_file) ([https://en.wikipedia.org/wiki/INI\\_file](https://en.wikipedia.org/wiki/INI_file)) format.
- (optionally) files containing data that cannot easily be represented by the INI format.

We have added the following conventions into the Rose configuration format:

1. The file name is normally called `rose*.conf`, e.g. `rose.conf` (page 198), `rose-app.conf` (page 194), `rose-meta.conf` (page 207), etc.
2. Only a hash # in the beginning of a line starts a comment. Empty lines and lines with only white spaces are ignored. There is no support for trailing comments. Comments are normally ignored when a configuration file is loaded. However, some comments are loaded if the following conditions are met:
  - Comment lines at the beginning of a configuration file up to but not including the 1st blank line or the 1st setting are comment lines associated with the file. They will re-appear at the top of the file when it is re-dumped.
  - Comment lines between a blank line and the next setting, and the comment lines between the previous setting and the next setting are comments associated with the next setting. The comment lines associated with a setting will re-appear before the setting when the file is re-dumped.
3. Only the equal sign = is used to delimit a key-value pair - because the colon : may be used in keys of namelist declarations.
4. A key-value pair declaration does not have to live under a section declaration. Such a declaration lives directly under the *root* level.
5. Key-value pair declarations following a line with only [ ] are placed directly under the root level.
6. Declarations are case sensitive. When dealing with case-insensitive inputs such as Fortran logicals or numbers in scientific notation, lowercase values should be used.
7. When writing namelist inputs, keys should be lowercase.
8. Declarations start at column 1. Continuations start at column >1.
  - Each line is stripped of leading and trailing spaces.
  - A newline \n character is prefixed to each continuation line.
  - If a continuation line has a leading equal sign = character, it is stripped from the line. This is useful for retaining leading white spaces in a continuation line.
9. A single exclamation ! or a double exclamation !! in front of a section (i.e. [ !SECTION]) or key=value pair (i.e. !key=value) denotes an ignored setting.
  - E.g. It will be ignored in run time but may be used by other Rose utilities.
  - A single exclamation denotes a user-ignored setting.
  - A double exclamation denotes a program-ignored setting. E.g. `rose config-edit` (page 250) may use a double exclamation to switch off a setting according to the setting metadata.
10. The open square bracket ([ ]) and close square bracket (]) characters cannot be used within a section declaration. E.g. [[hello], [hello]], [hello [world] and beyond] should all be errors on parsing.
11. If a section is declared twice in a file, the later section will append settings to the earlier one. If the same key in the same section is declared twice, the later value will override the earlier one. This logic applies to the state of a setting as well.
12. Once the file is parsed, declaration ordering is insignificant.

---

**Note:** Do not assume order of environment variables.

---

13. Values of settings accept syntax such as \$NAME or \${NAME} for environment variable substitution.

E.g.

```
# This is line 1 of the comment for this file.  
# This is line 2 of the comment for this file.  
  
# This comment will be ignored.  
  
# This is a comment for section-1.  
[section-1]  
# This is a comment for key-1.  
key-1=value 1  
# This comment will be ignored.  
  
# This is line 1 of the comment for key-2.  
# This is line 2 of the comment for key-2.  
key-2=value 2 line 1  
      value 2 line 2  
# This is a comment for key-3.  
key-3=value 3 line 1  
      =     value 3 line 2 has leading indentation.  
      =  
      =     value 3 line 3 is blank. This is line 4.  
  
# section-2 is user-ignored.  
[!section-2]  
key-4=value 4  
# ...  
  
[section-3]  
# key-5 is program ignored.  
!!key-5=value 5
```

---

**Note:** In this document, the shorthand SECTION=KEY=VALUE is used to represent a KEY=VALUE pair in a [SECTION] of an INI format file.

---

### 7.5.1 Goals

Suite configurations should be portable between users (at least at the same site). E.g.: another user should be able to run the same suite:

- without making ANY changes to it.
- without having to add/modify things in their \$HOME/.profile.

Input configurations should be programming language neutral.

- Any processing logic should be application/version independent, generic and future-proof.
- Data structure should be represented in formats easily understood and manipulatable by a human and a computer.

The life cycles of application configurations in a suite may differ from that of the suite.

- The configuration of an application may be independent of the suite.
- The configuration of an application should be portable between suitable suites.

The configurations are independent of the utilities. For example, the configuration metadata for the suite and application configurations will drive the Rose config editor GUI, but will not be bound or restricted by it.

## 7.5.2 Optional Configuration

In a Rose configuration directory, we can add an `opt` / sub-directory for optional configuration files. Optional configuration files contain additional configuration, which can be selected at run time to override the configuration in the main `rose-${TYPE}.conf` file. The name of each optional configuration should follow the syntax `rose-${TYPE}-${KEY}.conf`, where `${KEY}` is a short name to describe the override functionality of the optional configuration file.

A root level `opts=KEY ...` setting in the main configuration will tell the run time program to load the relevant optional configurations in the `opt` / sub-directory at run time. Individual Rose utilities may also read optional configuration keys from environment variables and/or command line options.

Where multiple `$KEY` settings are given, the optional configurations are applied in that order - for example, a setting:

```
opts=ketchup mayonnaise
```

implies loading the optional configuration `rose-app-ketchup.conf` and then the optional configuration `rose-app-mayonnaise.conf`, which may override the previous one.

By default, a Rose command will fail if an optional configuration file is missing. However, if you put the optional configuration key in brackets, then the optional configuration file is allowed to be missing. E.g.:

```
opts=ketchup (mayonnaise)
```

In the above example, `rose-app-mayonnaise.conf` can be missing.

Some Rose utilities (e.g. [rose suite-run](#) (page 263), [rose task-run](#) (page 267), [rose app-run](#) (page 245), etc) allow optional configurations to be selected at run time using:

1. The `ROSE_APP_OPT_CONF_KEYS` Environment variables.
2. The command line options `--opt-conf-key=KEY` or `-O KEY`.

---

**Tip:** See reference of individual commands for detail.

---

**Note:** By default optional configurations must exist else an error will be raised. To specify an optional configuration which may be missing write the name of the configuration inside parenthesis (e.g. `(foo)`).

---

## Optional Configurations and Metadata

Metadata utilities such as [rose app-upgrade](#) (page 246) and [rose macro](#) (page 254) treat each main + optional configuration as a separate entity to be transformed, upgraded, or validated. Use cases with more than one optional configuration are not handled.

When transforming or upgrading, each optional configuration is treated separately and re-created after the transform as a functional difference from the main upgraded configuration.

The logic for transforming or upgrading a main configuration `C` with optional configurations `O1` and `O2` into a new main configuration `Ct` and new optional configurations `O1t` and `O2t` can be represented like this:

```
C => Ct
C + O1 => C1t
C + O2 => C2t
O1t = C1t - Ct
O2t = C2t - Ct
```

### 7.5.3 Import Configuration

A root level `import=PATH1 PATH2...` setting in the main configuration will tell Rose utilities to search for configurations at PATH1, PATH2 (and so on) and inherit configuration and files from them if found.

---

**Tip:** At the moment, use of this is only encouraged for configuration metadata.

---

### 7.5.4 Re-define Configuration at Run Time

Some Rose utilities (e.g. `rose suite-run` (page 263), `rose task-run` (page 267), `rose app-run` (page 245), etc) allow you to re-define configuration settings at run time using the `--define=[SECTION]NAME=VALUE` or `-D [SECTION]NAME=VALUE` options on the command line. This would add new settings or override any settings defined in the main and optional configurations. E.g.:

```
# Set [env]FOO=foo, and [env]BAR=bar
# (Overriding any original settings of [env]FOO or [env]BAR)
rose task-run -D '[env]FOO=foo' -D '[env]BAR=bar'

# Switch off [env]BAZ
rose task-run -D '[env]!BAZ='
```

## 7.6 File Creation Mode

The application launcher will use the following logic to determine the root directory to install file targets with a relative path:

1. If the setting `rose-app.conf/file-install-root=PATH` (page 194) is specified in the application configuration file, its value will be used.
2. If the environment variable `ROSE_FILE_INSTALL_ROOT` (page 276) is specified, its value will be used.
3. Otherwise, the working directory of the task will be used.

**Rose Configuration \***

#### Config schemes

##### Options

- **fs** – The file system scheme. If a URI looks like an existing path in the file system, this scheme will be used.
- **namelist** – The namelist scheme. Refer to `namelist:NAME` sections in the configuration file.
- **svn** – The Subversion scheme. The location is a Subversion URL or an FCM location keyword. A URI with these schemes `svn`, `svn+ssh` and `fcm` are automatically recognised.
- **rsync** – This scheme is useful for pulling a file or directory from a remote host using `rsync` via `ssh`. A URI should have the form `HOST:PATH`.

Rose will automatically attempt to detect the type of a source (i.e. file, directory, URL), however, the name of the source can sometimes be ambiguous. E.g. A URL with a `http` scheme can be a path in a version control system, or a path to a plain file. The `schemes` (page 226) setting can be used to help the system to do the right thing. The syntax is as follows:

```
schemes=PATTERN-1=SCHEME-1
      =PATTERN-2=SCHEME-2
```

For example:

```
schemes=hpc*:*=rsync
      =http://host/svn-repos/*=svn

[file:foo.txt]
source=hpc1:/path/to/foo.txt

[file:bar.txt]
source=http://host/svn-repos/path/to/bar.txt
```

In the above example, a URI matching the pattern `hpc*:*` would use the `rsync` scheme to pull the source to the current host, and a URI matching the pattern `http://host/svn-repos/*` would use the `svn` scheme. For all other URIs, the system will try to make an intelligent guess.

---

**Note:** The system will always match a URI in the order as specified by the setting to avoid ambiguity.

---

**Note:** If the `rsync` scheme is used you can use the `User` ([http://man.openbsd.org/ssh\\_config#User](http://man.openbsd.org/ssh_config#User)) setting in `~/.ssh/config` to specify the user ID for logging into HOST if required.

---

#### **Config [file:TARGET]**

##### **Config source= SOURCE (alternate: source=(SOURCE))**

A space delimited list of sources for generating this file. A source can be the path to a regular file or directory in the file system (globbing is also supported - e.g. using "`\*.conf`" to mean all `.conf` files), or it may be a URI to a resource. If a source is a URI, it may point to a section with a supported scheme in the current configuration, e.g. a `namelist:NAME` section. Otherwise the URI must be in a supported scheme or be given sufficient information for the system to determine its scheme, e.g. via the `* /schemes` (page 226) setting.

---

**Tip:** Normally, a source that does not exist would trigger an error in run time. However, it may be useful to have an optional source for a file sometimes. In which case, the syntax `source=(SOURCE)` (page 227) can be used to specify an optional source. E.g. `source=namelist:foo (namelist:bar)` would allow `namelist:bar` to be missing or ignored without an error.

---

#### **Config checksum**

The expected MD5 checksum of the target. If specified, the file generation will fail if the actual checksum of the target does not match with this setting. This setting is only meaningful if TARGET is a regular file or a symbolic link to a regular file.

---

**Note:** An empty value for checksum tells the system to report the target checksum in verbose mode.

---

#### **Config mode**

**Default** auto

##### **Options**

- **auto** – Automatically determine action based on the value of `source` (page 227).
  - `source=`(page 227) - If source is undefined create an empty file.
  - `source=path` (page 227) - If source is a single path to a file or directory then the path will be coppied to the target path.
  - `source=file1 file2 ...` (page 227) - If the source is a list of files then the files will be concatenated in the target path.

- `source=dir1 dir2 ...` (page 227) - If the source is a list of directories then the directories will be transferred to the target path using `rsync`.
- **`mkdir`** – Creates an empty directory (`source` (page 227) must be a single path).
- **`symlink`** – Creates a symlink to the provided source, the source *does not* have to exist when the symlink is created (`source` (page 227) must be a single path).
- **`symlink+`** – Creates a symlink to the provided source, the source *must* exist when the symlink is created (`source` (page 227) must be a single path).

## 7.7 Rose Configuration API

### 7.7.1 CLI

The `rose config` (page 247) command provides a command line tool for reading, processing and dumping files written in the *Rose Configuration Format* (page 223):

```
$ echo -e "  
> [foo]  
> bar=baz  
> " > rose.conf  
  
$ rose config foo --file rose.conf  
bar=baz
```

For more information see the `rose config` (page 247) command line reference.

### 7.7.2 Python

Rose provides a Python API for loading, processing, editing and dumping Rose configurations via the `rose.config` (page 228) module located within the Rose Python library.

Simple INI-style configuration data model, loader and dumper.

**Synopsis:**

```
>>> # Create a config file.  
>>> with open('config.conf', 'w+') as config_file:  
...     config_file.write(''  
...     [foo]  
...     bar=Bar  
...     !baz=Baz  
...     '')
```

```
>>> # Load in a config file.  
>>> try:  
...     config_node = load('config.conf')  
... except ConfigSyntaxError:  
...     # Handle exception.  
...     pass
```

```
>>> # Retrieve config settings.  
>>> config_node.get(['foo', 'bar'])  
{'state': '', 'comments': [], 'value': 'Bar'}
```

```
>>> # Set new config settings.  
>>> _ = config_node.set(['foo', 'new'], 'New')
```

```
>>> # Overwrite existing config settings.
>>> _ = config_node.set(['foo', 'baz'], state=ConfigNode.STATE_NORMAL,
...                               value='NewBaz')
```

```
>>> # Write out config to a file.
>>> dump(config_node, sys.stdout)
[foo]
bar=Bar
baz=NewBaz
new>New
```

**Classes:**

<code>rose.config.ConfigNode</code> (page 232)([value, state, comments])	Represent a node in a configuration file.
<code>rose.config.ConfigNodeDiff</code> (page 237)()	Represent differences between two ConfigNode instances.
<code>rose.config.ConfigDumper</code> (page 229)([char_assign])	Dumper of a ConfigNode object in Rose INI format.
<code>rose.config.ConfigLoader</code> (page 230)([char_assign, ...])	Loader of an INI format configuration into a ConfigNode object.

**Functions:**

<code>rose.config.load</code> (page 243)(source[, root])	Shorthand for <code>ConfigLoader.load()</code> (page 230).
<code>rose.config.dump</code> (page 243)(root[, target, ...])	Shorthand for <code>ConfigDumper.dump()</code> (page 230).

**Limitations:**

- The loader does not handle trailing comments.

**What about the standard library ConfigParser? Well, it is problematic:**

- The comment character and style is hard-coded.
- The assignment character is hard-coded.
- A duplicated section header causes an exception to be raised.
- Option keys are transformed to lower case by default.
- It is far too complicated and confusing.

`class rose.config.ConfigDumper(char_assign='=')`  
Dumper of a ConfigNode object in Rose INI format.

**Examples**

```
>>> config_node = ConfigNode()
>>> _ = config_node.set(keys=['foo', 'bar'], value='Bar')
>>> _ = config_node.set(keys=['foo', 'baz'], value='Baz',
...                               comments=['Currently ignored!'],
...                               state=ConfigNode.STATE_USER_IGNORED)
>>> dumper = ConfigDumper()
>>> dumper(config_node, sys.stdout)
[foo]
bar=Bar
```

(continues on next page)

(continued from previous page)

```
#Currently ignored!
!baz=Baz
```

**dump**(*root*, *target*=<open file '<stdout>', *mode* 'w'>, *sort\_sections*=None, *sort\_option\_items*=None, *env\_escape\_ok*=False, *concat\_mode*=False)  
Format a ConfigNode object and write result to target.

**Parameters**

- **root** ([ConfigNode](#) (page 232)) – The root config node.
- **target** (*str/file*) – An open file handle or a string containing a file path. If not specified, the result is written to sys.stdout.
- **sort\_sections** (*fcn – optional*) – An optional argument that should be a function for sorting a list of section keys.
- **sort\_option\_items** (*fcn – optional*) – An optional argument that should be a function for sorting a list of option (key, value) tuples in string values.
- **env\_escape\_ok** (*bool – optional*) – An optional argument to indicate that \$NAME and \${NAME} syntax in values should be escaped.
- **concat\_mode** (*bool – optional*) – Switch on concatenation mode. If True, add [] before root level options.

**class** `rose.config.ConfigLoader`(*char\_assign*='=', *char\_comment*='#')  
Loader of an INI format configuration into a ConfigNode object.

**Example**

```
>>> with open('config.conf', 'w+') as config_file:
...     config_file.write('''
... [foo]
... !bar=Bar
... baz=Baz
... ''')
>>> loader = ConfigLoader()
>>> try:
...     config_node = loader.load('config.conf')
... except ConfigSyntaxError:
...     raise # Handle exception.
>>> config_node.get(keys=['foo', 'bar'])
{'state': '!', 'comments': [], 'value': 'Bar'}
```

**static can\_miss\_opt\_conf\_key**(*key*)  
Return KEY if key is a string like “(KEY)”, None otherwise.

**load**(*source*, *node*=None, *default\_comments*=None)  
Read a source configuration file.

**Parameters**

- **source** (*str*) – An open file handle or a string for a file path.
- **node** ([ConfigNode](#) (page 232)) – A ConfigNode object if specified, otherwise created.

**Returns** A new ConfigNode object.

**Return type** [ConfigNode](#) (page 232)

## Examples

```
>>> # Create example config file.
>>> with open('config.conf', 'w+') as config_file:
...     config_file.write('''
... [foo]
... # Some comment
... !bar=Bar
... ''')
```

```
>>> # Load config file.
>>> loader = ConfigLoader()
>>> try:
...     config_node = loader.load('config.conf')
... except ConfigSyntaxError:
...     raise # Handle exception.
>>> config_node.get(keys=['foo', 'bar'])
{'state': '!', 'comments': [' Some comment'], 'value': 'Bar'}
```

**load\_with\_opts** (*source*, *node=None*, *more\_keys=None*, *used\_keys=None*, *return\_config\_map=False*, *mark\_opt\_confs=False*)  
Read a source configuration file with optional configurations.

### Parameters

- **source** (*str*) – A file path.
- **node** (*ConfigNode* – *optional*) – A ConfigNode object if specified, otherwise one is created.
- **more\_keys** (*list* – *optional*) – A list of additional optional configuration names. If source is “rose-\${TYPE}.conf”, the file of each name should be “opt/rose-\${TYPE}-\${NAME}.conf”.
- **used\_keys** (*list* – *optional*) – If defined, it should be a list for this method to append to. The key of each successfully loaded optional configuration will be appended to the list (unless the key is already in the list). Missing optional configurations that are specified in more\_keys will not raise an error. If not defined, any missing optional configuration will trigger an OSError.
- **mark\_opt\_configs** (*bool* – *optional*) – if True, add comments above any settings which have been loaded from an optional config.
- **return\_config\_map** (*bool* – *optional*) – If True, construct and return a dict (config\_map) containing config names vs their uncombined nodes. Optional configurations use their opt keys as keys, and the main configuration uses ‘None’.

### Returns

**node or (node, config\_map):**

- node - The loaded configuration as a ConfigNode.
- config\_map - A dictionary containing opt\_conf\_key: ConfigNode pairs. Only returned if return\_config\_map is True.

**Return type** tuple

## Examples

```
>>> # Write config file.
>>> with open('config.conf', 'w+') as config_file:
...     config_file.write(''
```

(continues on next page)

(continued from previous page)

```

... [foo]
... bar=Bar
...
'')
>>> # Write optional config file (foo).
>>> os.mkdir('opt')
>>> with open('opt/config-foo.conf', 'w+') as opt_config_file:
...     opt_config_file.write('')
... [foo]
... bar=Baz
...
'''')

```

```

>>> loader = ConfigLoader()
>>> config_node, config_map = loader.load_with_opts(
...     'config.conf', more_keys=['foo'], return_config_map=True)
>>> config_node.get_value(keys=['foo', 'bar'])
'Baz'

```

```

>>> original_config_node = config_map[None]
>>> original_config_node.get_value(keys=['foo', 'bar'])
'Bar'

```

```

>>> optional_config_node = config_map['foo']
>>> optional_config_node.get_value(keys=['foo', 'bar'])
'Baz'

```

**class** rose.config.ConfigNode (*value=None*, *state=”*, *comments=None*)  
Represent a node in a configuration file.

**Nodes are stored hierarchically, for instance the following config** [foo] bar = Bar

When loaded by ConfigNode.load(file) would result in three levels of ConfigNodes, the first representing “root” (i.e. the top level of the config), one representing the config section “foo” and one representing the setting “bar”.

## Examples

```

>>> # Create a new ConfigNode.
>>> config_node = ConfigNode()

```

```

>>> # Add sub-nodes.
>>> _ = config_node.set(keys=['foo', 'bar'], value='Bar')
>>> _ = config_node.set(keys=['foo', 'baz'], value='Baz')
>>> config_node
{'state': '', 'comments': [],
 'value': {'foo': {'state': '', 'comments': [],
                  'value': {'baz': {'state': '', 'comments': [],
                                    'value': 'Baz'},
                            'bar': {'state': '', 'comments': [],
                                    'value': 'Bar'}}}}}

```

```

>>> # Set the state of a node.
>>> _ = config_node.set(keys=['foo', 'bar'],
...                      state=ConfigNode.STATE_USER_IGNORED)

```

```

>>> # Get the value of the node at a position.
>>> config_node.get_value(keys=['foo', 'baz'])
'Baz'

```

```
>>> # Walk over the hierarchical structure of a node.
>>> [keys for keys, sub_node in config_node.walk()]
[['foo'], ['foo', 'bar'], ['foo', 'baz']]
```

```
>>> # Walk over the config skipping ignored sections.
>>> [keys for keys, sub_node in config_node.walk(no_ignore=True)]
[['foo'], ['foo', 'baz']]
```

```
>>> # Add two ConfigNode instances to create a new "merged" node.
>>> another_config_node = ConfigNode()
>>> _ = another_config_node.set(keys=['new'], value='New')
>>> new_config_node = config_node + another_config_node
>>> [keys for keys, sub_node in new_config_node.walk()]
[['foo'], ['foo', 'baz'], ['foo', 'bar'], ['', 'new']]
```

**STATE\_NORMAL** = ''

The default state of a ConfigNode.

**STATE\_SYS\_T\_IGNORED** = '!!!'

ConfigNode state if a metadata operation has logically ignored the config.

**STATE\_USER\_IGNORED** = '!'

ConfigNode state if it has been specifically ignored in the config.

**add**(*config\_diff*)

Apply a ConfigNodeDiff object to self.

**Parameters** **config\_diff** ([ConfigNodeDiff](#) (page 237)) – A diff to apply to this ConfigNode.

**Examples**

```
>>> # Create ConfigNode
>>> config_node = ConfigNode()
>>> _ = config_node.set(keys=['foo', 'bar'], value='Bar')
>>> [keys for keys, sub_node in config_node.walk()]
[['foo'], ['foo', 'bar']]
```

```
>>> # Create ConfigNodeDiff
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(keys=['foo', 'baz'],
...   data='Baz')
```

```
>>> # Apply ConfigNodeDiff to ConfigNode
>>> config_node.add(config_node_diff)
>>> [keys for keys, sub_node in config_node.walk()]
[['foo'], ['foo', 'bar'], ['foo', 'baz']]
```

**get**(*keys=None, no\_ignore=False*)

Return a node at the position of keys, if any.

**Parameters**

- **keys** (*list, optional*) – A list defining a hierarchy of node.value ‘keys’. If an entry in keys is the null string, it is skipped.
- **no\_ignore** (*bool, optional*) – If True any ignored nodes will not be returned.

**Returns** The config node located at the position of keys or None.

**Return type** [ConfigNode](#) (page 232)

## Examples

```
>>> # Create ConfigNode.
>>> config_node = ConfigNode()
>>> _ = config_node.set(['foo', 'bar'], 'Bar')
>>> _ = config_node.set(['foo', 'baz'], 'Baz',
...     state=ConfigNode.STATE_USER_IGNORED)
```

```
>>> # A ConfigNode containing sub-nodes.
>>> config_node.get(keys=['foo'])
{'state': '', 'comments': [], 'value': {'baz': {'state': '!', 'comments': [], 'value': 'Baz'}, 'bar': {'state': '', 'comments': [], 'value': 'Bar'}}}
```

```
>>> # A bottom level sub-node.
>>> config_node.get(keys=['foo', 'bar'])
{'state': '', 'comments': [], 'value': 'Bar'}
```

```
>>> # Skip ignored nodes.
>>> print config_node.get(keys=['foo', 'baz'], no_ignore=True)
None
```

### `get_filter(no_ignore)`

Return this ConfigNode unless no\_ignore and node is ignored.

**Parameters** `no_ignore (bool)` – If True only return this node if it is not ignored.

**Returns** This ConfigNode unless no\_ignore and node is ignored.

**Return type** `ConfigNode` (page 232)

## Examples

```
>>> config_node = ConfigNode(value=42)
>>> config_node.get_filter(False)
{'state': '', 'comments': [], 'value': 42}
>>> config_node.get_filter(True)
{'state': '', 'comments': [], 'value': 42}
```

```
>>> config_node = ConfigNode(value=42,
...     state=ConfigNode.STATE_USER_IGNORED)
>>> config_node.get_filter(False)
{'state': '!', 'comments': [], 'value': 42}
>>> print config_node.get_filter(True)
None
```

### `get_value(keys=None, default=None)`

Return the value of a normal node at the position of keys, if any.

If the node does not exist or is ignored, return None.

#### Parameters

- `keys (list, optional)` – A list defining a hierarchy of node.value ‘keys’. If an entry in keys is the null string, it is skipped.
- `default (obj, optional)` – Return default if the value is not set.

**Returns** The value of this ConfigNode at the position of keys or default if not set.

**Return type** obj

## Examples

```
>>> # Create ConfigNode.
>>> config_node = ConfigNode(value=42)
>>> _ = config_node.set(['foo'], 'foo')
>>> _ = config_node.set(['foo', 'bar'], 'Bar')
```

```
>>> # Get value without specifying keys returns the value of the
>>> # root ConfigNode (which in this case is a dict of its
>>> # sub-nodes).
>>> config_node.get_value()
{'foo': {'state': '', 'comments': [],
         'value': {'bar': {'state': '', 'comments': [],
                           'value': 'Bar'}}}}
```

```
>>> # Intermediate level ConfigNode.
>>> config_node.get_value(keys=['foo'])
{'bar': {'state': '', 'comments': [], 'value': 'Bar'}}
```

```
>>> # Bottom level ConfigNode.
>>> config_node.get_value(keys=['foo', 'bar'])
'Bar'
```

```
>>> # If there is no node located at the position of keys or if
>>> # that node is unset then the default value is returned.
>>> config_node.get_value(keys=['foo', 'bar', 'baz'],
...                         default=True)
True
```

### `is_ignored()`

Return True if current node is in the “ignored” state.

### `set(keys=None, value=None, state=None, comments=None)`

Set node properties at the position of keys, if any.

#### Parameters

- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. If an entry in keys is the null string, it is skipped.
- **value** (*obj*) – The node.value property to set at this position.
- **state** (*str*) – The node.state property to set at this position. If None, the node.state property is unchanged.
- **comments** (*str*) – The node.comments property to set at this position. If None, the node.comments property is unchanged.

**Returns** This config node.

**Return type** *ConfigNode* (page 232)

## Examples

```
>>> # Create ConfigNode.
>>> config_node = ConfigNode()
>>> config_node
{'state': '', 'comments': [], 'value': {}}
```

```
>>> # Add a sub-node at the position 'foo' with the comment 'Info'.
>>> config_node.set(keys=['foo'], comments='Info')
...
{'state': '', 'comments': [], 'value': {'foo': {'state': '', 'comments': 'Info', 'value': {}}}}
```

```
>>> # Set the value for the sub-node at the position
>>> # 'foo' to 'Foo'.
>>> config_node.set(keys=['foo'], value='Foo')
...
{'state': '', 'comments': [], 'value': {'foo': {'state': '', 'comments': 'Info', 'value': 'Foo'}}}
```

```
>>> # Set the value of the ConfigNode to True, this overwrites all
>>> # sub-nodes!
>>> config_node.set(keys=[''], value=True)
{'state': '', 'comments': [], 'value': True}
```

**unset**(*keys=None*)

Remove a node at the position of *keys*, if any.

**Parameters** **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. If an entry in *keys* is the null string, it is skipped.

**Returns** The ConfigNode instance that was removed, else None.

**Return type** *ConfigNode* (page 232)

**Examples**

```
>>> # Create ConfigNode.
>>> config_node = ConfigNode()
>>> _ = config_node.set(keys=['foo'], value='Foo')
```

```
>>> # Unset without providing any keys does nothing.
>>> print config_node.unset()
None
```

```
>>> # Unset with invalid keys does nothing.
>>> print config_node.unset(keys=['bar'])
None
```

```
>>> # Unset with valid keys removes the node from the node.
>>> config_node.unset(keys=['foo'])
{'state': '', 'comments': [], 'value': 'Foo'}
```

```
>>> config_node
{'state': '', 'comments': [], 'value': {}}
```

**walk**(*keys=None, no\_ignore=False*)

Return all keylist - sub-node pairs below *keys*.

**Parameters**

- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. If an entry in *keys* is the null string, it is skipped.
- **no\_ignore** (*bool*) – If True any ignored nodes will be skipped.

**Yields**

**tuple - (keys, sub\_node)**

- keys (list) - A list defining a hierarchy of node.value ‘keys’. If a sub-node is at the top level, and does not contain any node children, a null string will be prepended to the returned keylist.
- sub\_node (ConfigNode) - The config node at the position of keys.

**Examples**

```
>>> config_node = ConfigNode()
>>> _ = config_node.set(['foo', 'bar'], 'Bar')
>>> _ = config_node.set(['foo', 'baz'], 'Baz',
...                         state=ConfigNode.STATE_USER_IGNORED)
```

```
>>> # Walk over the full hierarchy.
>>> [keys for keys, sub_node in config_node.walk()]
[['foo'], ['foo', 'bar'], ['foo', 'baz']]
```

```
>>> # Walk over one branch of the hierarchy
>>> [keys for keys, sub_node in config_node.walk(keys=['foo'])]
[['foo', 'bar'], ['foo', 'baz']]
```

```
>>> # Skip over ignored nodes.
>>> [keys for keys, sub_node in config_node.walk(no_ignore=True)]
[['foo'], ['foo', 'bar']]
```

```
>>> # Invalid/non-existent keys.
>>> [keys for keys, sub_node in config_node.walk(
...     keys=['elephant'])]
[]
```

**class** rose.config.ConfigNodeDiff  
Represent differences between two ConfigNode instances.

**Examples**

```
>>> # Create a new ConfigNodeDiff.
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(keys=['bar'],
...   data=('Bar', None, None,))
```

```
>>> # Create a new ConfigNode.
>>> config_node = ConfigNode()
>>> _ = config_node.set(keys=['baz'], value='Baz')
```

```
>>> # Apply the diff to the node.
>>> config_node.add(config_node_diff)
>>> [(keys, sub_node.get_value()) for keys, sub_node in
...     config_node.walk()]
[(['', 'baz'], 'Baz'), (['', 'bar'], 'Bar')]
```

```
>>> # Create a ConfigNodeDiff by comparing two ConfigNodes.
>>> another_config_node = ConfigNode()
>>> _ = another_config_node.set(keys=['bar'], value='NewBar')
>>> _ = another_config_node.set(keys=['new'], value='New')
```

(continues on next page)

(continued from previous page)

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_from_configs(config_node,
...   another_config_node)
>>> config_node_diff.get_added()
[((', 'new'), ('New', '', []))]
>>> config_node_diff.get_removed()
[((', 'baz'), ('Baz', '', []))]
>>> config_node_diff.get_modified()
[((', 'bar'), (('Bar', '', []), ('NewBar', '', [])))]
```

```
>>> # Inverse a ConfigNodeDiff.
>>> reversed_diff = config_node_diff.get_reversed()
>>> reversed_diff.get_added()
[((', 'baz'), ('Baz', '', []))]
```

**delete\_removed()**

Deletes all ‘removed’ keys from this ConfigNodeDiff.

**Examples**

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_removed_setting(['foo'],
...   ('foo', None, None))
>>> config_node_diff.delete_removed()
>>> config_node_diff.get_removed()
[]
```

**get\_added()**

Return a list of tuples of added keys with their data.

The data is a tuple of value, state, comments, where value is set to None for sections.

**Returns**

A list of the form [(keys, data), ...]

- keys - The position of an added setting.
- data - Tuple of the form (value, state, comments) of the properties of the added setting.

**Return type** list**Examples**

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(['foo'],
...   ('Foo', None, None))
>>> config_node_diff.get_added()
[('foo', ('Foo', None, None))]
```

**get\_all\_keys()**

Return all keys affected by this ConfigNodeDiff.

**Returns** A list containing any keys affected by this diff as tuples, e.g. ('foo', 'bar').

**Return type** list

## Examples

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(['foo'], ('foo', None, None))
>>> config_node_diff.set_removed_setting(['bar'],
...   ('bar', None, None))
>>> config_node_diff.get_all_keys()
[('bar',), ('foo',)]
```

### `get_as_opt_config()`

Return a ConfigNode such that main + new\_node = main + diff.

Add all the added settings, add all the modified settings, add all the removed settings as user-ignored.

**Returns** A new ConfigNode instance.

**Return type** `ConfigNode` (page 232)

## Example

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(['foo'],
...   ('Foo', None, None))
>>> config_node_diff.set_removed_setting(['bar'],
...   ('Bar', None, None))
>>> config_node = config_node_diff.get_as_opt_config()
>>> list(config_node.walk())
[(['', 'bar'], {'state': '!!', 'comments': [], 'value': 'Bar'}),
 [('', 'foo'), {'state': '', 'comments': [], 'value': 'Foo'}]]
```

### `get_modified()`

Return a dict of altered keys with before and after data.

The data is a list of two tuples (before and after) of value, state, comments, where value is set to None for sections.

**Returns**

A list of the form [(keys, data), ...]:

- keys - The position of an added setting.
- data - Tuple of the form (value, state, comments) for the properties of the setting before the modification.
- old\_data - The same tuple as data but representing the properties of the setting after the modification.

**Return type** list

## Examples

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_modified_setting(
...     ['foo'], ('Foo', None, None), ('New Foo', None, None))
>>> config_node_diff.get_modified()
[((('foo',), ((('foo', None, None), ('New Foo', None, None))))]
```

### `get_removed()`

Return a dict of removed keys with their data.

The data is a tuple of value, state, comments, where value is set to None for sections.

**Returns**

- **keys** - The position of an added setting.
- **data** - Tuple of the form (value, state, comments) of the properties of the removed setting.

**Return type** list - A list of the form [(keys, data), ..]

**Examples**

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_removed_setting(keys=['foo'],
...   data=('foo', None, None))
>>> config_node_diff.get_removed()
[('foo',), ('foo', None, None)]
```

**get\_reversed()**

Return an inverse (add->remove, etc) copy of this ConfigNodeDiff.

**Returns** A new ConfigNodeDiff instance.

**Return type** *ConfigNodeDiff* (page 237)

**Examples**

```
>>> # Create ConfigNodeDiff instance.
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(['foo'], ('foo', None, None))
>>> config_node_diff.set_removed_setting(['bar'],
...   ('bar', None, None))
... 
```

```
>>> # Generate reversed diff.
>>> reversed_diff = config_node_diff.get_reversed()
>>> reversed_diff.get_added()
[('bar',), ('bar', None, None)]
>>> reversed_diff.get_removed()
[('foo',), ('foo', None, None)]
```

**set\_added\_setting(*keys, data*)**

Set a config setting to be “added” in this ConfigNodeDiff.

**Parameters**

- **keys** (*list/tuple*) – The position of the setting to add.
- **data** (*obj, str, str*) – A tuple (value, state, comments) for the setting to add.

**Examples**

```
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_added_setting(['foo'],
...   ('Foo', None, None, ))
... 
>>> config_node_diff.set_added_setting(
...     ['bar'],
...     ('Bar', ConfigNode.STATE_USER_IGNORED, 'Some Info', ))
```

```
>>> config_node = ConfigNode()
>>> config_node.add(config_node_diff)
```

```
>>> list(config_node.walk())
[(['', 'bar'], {'state': '!', 'comments': 'Some Info',
               'value': 'Bar'}),
 ('', 'foo'), {'state': '', 'comments': [], 'value': 'Foo'})]
```

**set\_from\_configs**(*config\_node\_1*, *config\_node\_2*)

Create diff data from two ConfigNode instances.

**Parameters**

- **config\_node\_1** ([ConfigNode](#) (page 232)) – The node for which to base the diff off of.
- **config\_node\_2** ([ConfigNode](#) (page 232)) – The “new” node the changes of which this diff will “apply”.

**Example**

```
>>> # Create two ConfigNode instances to compare.
>>> config_node_1 = ConfigNode()
>>> _ = config_node_1.set(keys=['foo'])
>>> config_node_2 = ConfigNode()
>>> _ = config_node_2.set(keys=['bar'])
```

```
>>> # Create a ConfigNodeDiff instance.
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_from_configs(config_node_1, config_node_2)
>>> config_node_diff.get_added()
[('bar',), (None, '', [])]
>>> config_node_diff.get_removed()
[('foo',), (None, '', [])]
```

**set\_modified\_setting**(*keys*, *old\_data*, *data*)

Set a config setting to be “modified” in this ConfigNodeDiff.

If a property in both the old\_data and data (new data) are both set to None then no change will be made to any pre-existing value.

**Parameters**

- **keys** (*list/tuple*) – The position of the setting to add.
- **old\_data** (*obj, str, str*) – A tuple (value, state, comments) for the “current” properties of the setting to modify.
- **data** (*obj, str, str*) – A tuple (value, state, comments) for “new” properties to change this setting to.

**Examples**

```
>>> # Create a ConfigNodeDiff.
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_modified_setting(
...     ['foo'], ('Foo', None, None), ('New Foo', None, None))
```

```
>>> # Create a ConfigNode.
>>> config_node = ConfigNode()
>>> _ = config_node.set(keys=['foo'], value='Foo',
...                      comments='Some Info')
```

```
>>> # Apply the ConfigNodeDiff to the ConfigNode
>>> config_node.add(config_node_diff)
>>> config_node.get(keys=['foo'])
{'state': '', 'comments': 'Some Info', 'value': 'New Foo'}
```

**set\_removed\_setting**(*keys*, *data*)  
Set a config setting to be “removed” in this ConfigNodeDiff.

#### Parameters

- **keys** (*list*) – The position of the setting to add.
- **data** (*obj*, *str*, *str*) – A tuple (value, state, comments) of the properties for the setting to remove.

#### Example

```
>>> # Create a ConfigNodeDiff.
>>> config_node_diff = ConfigNodeDiff()
>>> config_node_diff.set_removed_setting(['foo'], ('X', 'Y', 'Z'))
```

```
>>> # Create a ConfigNode.
>>> config_node = ConfigNode()
>>> _ = config_node.set(keys=['foo'], value='Foo',
...                      comments='Some Info')
```

```
>>> # Apply the ConfigNodeDiff to the ConfigNode
>>> config_node.add(config_node_diff)
>>> print config_node.get(keys=['foo'])
None
```

**exception** `rose.config.ConfigSyntaxError`(*code*, *file\_name*, *line\_num*, *col\_num*, *line*)

Exception raised for syntax error loading a configuration file.

##### `exc.code`

Error code. Can be one of: ConfigSyntaxError.BAD\_CHAR (bad characters in a name) ConfigSyntaxError.BAD\_SYNTAX (general syntax error)

##### `exc.file_name`

The name of the file that triggers the error.

##### `exc.line_num`

The line number (from 1) in the file with the error.

##### `exc.col_num`

The column number (from 0) in the line with the error.

##### `exc.line`

The content of the line that contains the error.

#### Examples

```
>>> with open('config.conf', 'w+') as config_file:
...     config_file.write('[foo][foo]')
>>> loader = ConfigLoader()
>>> try:
...     loader.load('config.conf')
... except ConfigSyntaxError as exc:
...     print 'Error (%s) in file "%s" at %s:%s' % (
```

(continues on next page)

(continued from previous page)

```
...           exc.code, exc.file_name, exc.line_num, exc.col_num)
Error (BAD_CHAR) in file "..." at 1:5
```

`rose.config.dump`(*root*, *target*=*open file '<stdout>'*, *mode*='w', *sort\_sections*=None,  
                  *sort\_option\_items*=None, *env\_escape\_ok*=False)  
Shorthand for [ConfigDumper.dump\(\)](#) (page 230).

`rose.config.load`(*source*, *root*=None)  
Shorthand for [ConfigLoader.load\(\)](#) (page 230).

`rose.config.sort_element`(*elem\_1*, *elem\_2*)  
Sort pieces of text, numerically if possible.

`rose.config.sort_settings`(*setting\_1*, *setting\_2*)  
Sort sections and options, by numeric element if possible.



## COMMAND REFERENCE

### 8.1 Rose Commands

#### 8.1.1 rose app-run

```
rose app-run [OPTIONS] [--] [COMMAND ...]
```

Run an application according to its configuration.

May run a builtin application (if the mode setting in the configuration specifies the name of a builtin application) or a command.

Determine the command to run in this order:

1. If COMMAND is specified, invoke the command.
2. If the --command-key=KEY option is defined, invoke the command specified in [command] KEY.
3. If the ROSE\_APP\_COMMAND\_KEY environment variable is set, the command specified in the [command] KEY setting in the application configuration whose KEY matches it is used.
4. If the environment variable ROSE\_TASK\_NAME is defined and a setting in the [command] section has a key matching the value of the environment variable, then the value of the setting is used as the command.
5. Invoke the command specified in [command] default.

#### OPTIONS

- app-mode=MODE Run a builtin application identified by MODE. The default MODE is command.
- config=DIR, -C DIR Specify the configuration directory of the application. (default=\$PWD)
- command-key=KEY, -c KEY Invoke the command specified in [command] KEY.
- define=[SECTION]KEY=VALUE, -D [SECTION]KEY=VALUE Each of these overrides the [SECTION]KEY setting with a given VALUE. Can be used to disable a setting using the syntax --define=[SECTION]!KEY or even --define=[!SECTION].
- install-only, -i Install files only. Do not invoke the run.
- new, -N Remove all items in \$PWD before doing anything. This option only works with the --config=DIR option and if \$PWD is not DIR.
- no-overwrite Do not overwrite existing files.
- opt-conf-key=KEY, -O KEY, --opt-conf-key='(KEY)', -O '(KEY)' Each of these switches on an optional configuration identified by KEY. The configurations are applied first-to-last. The (KEY) syntax denotes an optional configuration that can be missing. Otherwise, the optional configuration must exist.
- quiet, -q Decrement verbosity.
- verbose, -v Increment verbosity.

## ENVIRONMENT VARIABLES

**optional ROSE\_APP\_COMMAND\_KEY** Switch to a particular command specified in [command] KEY.

**optional ROSE\_APP\_OPT\_CONF\_KEYS** Each KEY in this space delimited list switches on an optional configuration. The configurations are applied first-to-last.

**optional ROSE\_FILE\_INSTALL\_ROOT** If specified, change to the specified directory to install files.

---

### 8.1.2 rose app-upgrade

```
rose app-upgrade [OPTIONS] [VERSION]
```

Upgrade an application configuration using metadata upgrade macros.

Alternatively, show the available upgrade/downgrade versions:

- = indicates the current version.
- \* indicates the default version to change to.

If an application contains optional configurations, loop through each one, combine with the main, upgrade it, and re-create it as a diff vs the upgraded main configuration.

#### OPTIONS

**--all-versions, -a** Use all tagged versions.

**--config=DIR, -C DIR** Use configuration in DIR instead of \$PWD.

**--downgrade, -d** Downgrade the version instead of upgrade.

**--meta-path=PATH, -M PATH** Prepend PATH to the metadata search path. (look here first). This option can be used repeatedly to load multiple paths.

**--non-interactive, --yes, -y** Switch off interactive prompting.

**--output=DIR, -O DIR** The location of the output directory.

**--quiet, -q** Reduce verbosity.

#### ARGUMENTS

**VERSION** A version to change to. If no version is specified, show available versions. If --non-interactive is used, use the latest version available. If --non-interactive and --downgrade are used, use the earliest version available.

## ENVIRONMENT VARIABLES

**optional ROSE\_META\_PATH** Prepend \$ROSE\_META\_PATH to the metadata search path.

---

### 8.1.3 rose bush

```
rose bush start [PORT] # start ad-hoc web service server (on PORT)
rose bush stop          # stop ad-hoc web service server
rose bush stop -y       # stop ad-hoc web service server w/o prompting
rose bush              # print status of ad-hoc web service server
```

Start/stop ad-hoc Rose Bush web service server.

For `rose bush start`, if PORT is not specified, use port 8080.

Rose Bush is a web service for browsing users' Rose suite logs via an HTTP interface.

#### OPTIONS

---

**--non-interactive, --yes, -y** (For stop only.) Switch off interactive prompting (=answer yes to everything)

**--service-root, -R** (For start only.) Include web service name under root of URL.

---

### 8.1.4 rose check-software

```
rose check-software [OPTIONS]
```

Check software dependencies for the rose documentation builder.

#### OPTIONS

---

### 8.1.5 rose config

```
# Print the value of OPTION in SECTION.
rose config SECTION OPTION

# Print the value of OPTION in SECTION in FILE.
rose config --file=FILE SECTION OPTION

# Print the value of OPTION in SECTION if exists, or VALUE otherwise.
rose config --default=VALUE SECTION OPTION

# Print the OPTION=VALUE pairs in SECTION.
rose config SECTION

# Print the value of a top level OPTION.
rose config OPTION

# Print the OPTION keys in SECTION.
rose config --keys SECTION

# Print the SECTION keys.
rose config --keys

# Exit with 0 if OPTION exists in SECTION, or 1 otherwise.
rose config -q SECTION OPTION

# Exit with 0 if SECTION exists, or 1 otherwise.
rose config -q SECTION

# Combine the configurations in FILE1 and FILE2, and dump the result.
rose config --file=FILE1 --file=FILE2

# Print the value of OPTION in SECTION of the metadata associated with
# the specified config FILE
rose config --file=FILE --meta SECTION OPTION

# Print the value of a specified metadata KEY
rose config --meta-key=KEY
```

Parse and print rose configuration files.

With no option and no argument, print the Rose site + user configuration.

#### OPTIONS

---

**--default** Specify a default value.

**--env-var-process, -E** Process environment variable substitution. (Only works when returning a string value.)

**--file=FILE, -f FILE** Each of these specifies a configuration file. If none specified, read from \$THIS/../../etc/rose.conf and \$HOME/.metomi/rose.conf (where \$THIS is the location of this command).

**--keys, -k** If specified, only print the SECTION keys in the configuration file or the OPTION keys in a SECTION.

**--meta** If specified, operate on the metadata associated with the configuration FILE.

**--meta-key=KEY** If specified, prints the value of a specified metadata flag KEY. Cannot be used in conjunction with --file=FILE.

**--no-opts** Do not load optional configurations.

**--print-conf** If specified, prints result as a Rose configuration file snippet. This allows the output to be concatenated into another Rose configuration file.

**--print-ignored, -i** If specified, the program will print ignored !OPTION=VALUE where relevant.

**--quiet, -q** Exit with 0 if the specified SECTION and/or OPTION exist in the configuration, or 1 otherwise.

## ENVIRONMENT VARIABLES

**optional ROSE\_META\_PATH** Prepend \$ROSE\_META\_PATH to the metadata search path.

---

### 8.1.6 rose config-diff

```
# Display the metadata-annotated diff between two Rose config files.
rose config-diff FILE1 FILE2

# Display the metadata-annotated diff between two Rose config dirs.
rose config-diff DIR1 DIR2

# Display the diff, ignoring particular setting patterns
rose config-diff --ignore=namelist:foo FILE1 FILE2

# Display the diff with a particular diff tool
rose config-diff --diff-tool=kdiff3 FILE1 FILE2

# Display the diff with some diff tool specific options/arguments
rose config-diff FILE1 FILE2 -- [DIFF_OPTIONS] [DIFF_ARGUMENTS]
```

Display the metadata-annotated difference between two Rose config files.

## OPTIONS

**--diff-tool=TOOL** Specify an alternate diff tool like diffuse or kompare.

**--graphical, -g** Use a graphical diff tool.

**--ignore=PATTERN, -i PATTERN** Ignore settings that contain the regular expression PATTERN in their id. Can be specified more than once. PATTERN may also be a key used in site or user configuration which expands to a list of patterns. See CONFIGURATION below.

**--meta-path=PATH, -M PATH** Prepend PATH to the metadata search path (look here first). This option can be used repeatedly to load multiple paths.

**--opt-conf-key-1=KEY, --opt-conf-key-1='(KEY)'** opt-conf-key for the file in the first argument. Each of these switches on an optional configuration identified by KEY. The configurations are applied

first-to-last. The (KEY) syntax denotes an optional configuration that can be missing. Otherwise, the optional configuration must exist.

**--opt-conf-key-2=KEY, --opt-conf-key-2='(KEY)'** opt-conf-key for the file in the second argument. Each of these switches on an optional configuration identified by KEY. The configurations are applied first-to-last. The (KEY) syntax denotes an optional configuration that can be missing. Otherwise, the optional configuration must exist.

**--properties=PROPERTIES, -p PROPERTIES** Only display these metadata properties. This should be a comma separated list of metadata options, such as title,description,help.

## ENVIRONMENT VARIABLES

**optional ROSE\_META\_PATH** Prepend \$ROSE\_META\_PATH to the metadata search path.

## ARGUMENTS

**PATH1, PATH2** Two Rose configuration files or directories to compare. If the path is a directory, look underneath for a Rose configuration file. ‘-‘ for PATH1 or PATH2 denotes read in from standard input.

-- Options and arguments after a -- token are passed directly to the diff tool.

## CONFIGURATION

**[external]diff-tool, [external]gdiff-tool** You can override the default non-graphical and graphical diff tools by setting e.g:

```
[external]
diff-tool=diff3
gdiff-tool=kompare
```

in your site or user Rose configuration (rose.conf).

**[rose-config-diff]properties, [rose-config-diff]ignore{...}** You can override the default metadata properties to display by setting e.g:

```
[rose-config-diff]
properties=title,ns,description,help
```

in your site or user Rose configuration (rose.conf). You can also set shorthand ignore patterns by setting e.g.:

```
[rose-config-diff]
ignore{foo}=namelist:bar,namelist:baz
```

in the same location. This will allow you to run:

```
rose config-diff --ignore=foo ...
```

instead of:

```
rose config-diff --ignore=namelist:bar --ignore=namelist:baz ...
```

### 8.1.7 rose config-dump

```
rose config-dump
rose config-dump -C /path/to/conf/dir
rose config-dump -f /path/to/file1 -f /path/to/file2
```

Re-dump Rose configuration files in the common format.

Load and dump "rose-\*.conf" files in place. Apply format-specific pretty-printing.

By default, it recursively loads and dumps all `rose-*.conf` files in the current working directory.

## OPTIONS

**--config=DIR, -C DIR** Change directory to DIR before looking for configuration files. (default=\$PWD)  
**--file=FILE, -f FILE** Each of these specifies a configuration file. If --config=DIR is specified and FILE is a relative path, FILE is assumed to be relative to DIR.  
**--no-pretty** Do not apply format-specific pretty-printing.  
**--quiet, -q** Decrement verbosity. Do not report modified files.

---

## 8.1.8 rose config-edit

```
rose config-edit [OPTIONS]... [PAGE_PATH]...
```

Launch the GTK+ GUI to edit a suite or application configuration.

If a suite contains more than 10 applications then they will only be loaded after startup, on demand.

## OPTIONS

**--config=DIR, -C DIR** A path to either:

1. a directory containing a suite with a file named `suite.rc` and a directory called `app` containing subdirectories with files named `rose-app.conf`, in the format specified in the Rose pages.
2. a directory containing a single ‘application’ - a file named `rose-app.conf` and an optional subdirectory called `file` with other application files.

**--load-all-apps** Force loading of all applications on startup.

**--load-no-apps** Load applications in the suite on demand.

**--meta-path=PATH, -M PATH** Prepend PATH to the search path for metadata (look here first). This option can be used repeatedly to load multiple paths.

**--new** Launch, ignoring any configuration.

**--no-metadata** Launch with metadata switched off.

**--no-warn=WARNING-TYPE** Suppress warnings of the provided type. WARNING-TYPE may be:

**version** Suppress “Could not find metadata for app/version, using app/HEAD” warnings.

## ARGUMENTS

**PAGE\_PATH** One or more paths to open on load, pages may be full or partial namespaces e.g. `foo/bar/env` or `env`.

---

**Note:** Opens the shortest namespace that matches the provided string.

---

## ENVIRONMENT VARIABLES

**optional ROSE\_META\_PATH** Prepend \$ROSE\_META\_PATH to the metadata search path.

---

## 8.1.9 rose date

```

# 1. Print date time point
# 1.1 Current date time with an optional offset
rose date [--offset=OFFSET]
rose date now [--offset=OFFSET]
rose date ref [--offset=OFFSET]
# 1.2 Task cycle date time with an optional offset
#     Assume: export ROSE_TASK_CYCLE_TIME=20371225T000000Z
rose date -c [--offset=OFFSET]
rose date -c ref [--offset=OFFSET]
# 1.3 A specific date time with an optional offset
rose date 20380119T031407Z [--offset=OFFSET]

# 2. Print duration
# 2.1 Between now (+ OFFSET1) and a future date time (+ OFFSET2)
rose date now [--offset1=OFFSET1] 20380119T031407Z [--offset2=OFFSET2]
# 2.2 Between a date time in the past and now
rose date 19700101T000000Z now
# 2.3 Between task cycle time (+ OFFSET1) and a future date time
#     Assume: export ROSE_TASK_CYCLE_TIME=20371225T000000Z
rose date -c ref [--offset1=OFFSET1] 20380119T031407Z
# 2.4 Between task cycle time and now (+ OFFSET2)
#     Assume: export ROSE_TASK_CYCLE_TIME=20371225T000000Z
rose date -c ref now [--offset2=OFFSET2]
# 2.5 Between a date time in the past and the task cycle date time
#     Assume: export ROSE_TASK_CYCLE_TIME=20371225T000000Z
rose date -c 19700101T000000Z ref
# 2.6 Between 2 specific date times
rose date 19700101T000000Z 20380119T031407Z

# 3. Convert ISO8601 duration
# 3.1 Into the total number of hours (H), minutes (M) or seconds (S)
#     it represents, preceed negative durations with a double backslash
#     (e.g. \-PT1H)
rose date --as-total=s PT1H

```

Parse and print 1. a date time point or 2. a duration.

1. With 0 or 1 argument. Print the current or the specified date time point with an optional offset.
2. With 2 arguments. Print the duration between the 2 arguments.

## OPTIONS

**--as-total=TIME\_FORMAT** Used to express an ISO8601 duration in the specified time format, hours H, minutes M or seconds S.

**--calendar=gregorian|360day|365day|366day** Specify the calendar mode. See CALENDAR MODE below.

**--offset1=OFFSET, --offset=OFFSET, -s OFFSET, -1 OFFSET** Specify 1 or more offsets to add to argument 1 or the current time. See OFFSET FORMAT below.

**--offset2=OFFSET, -2 OFFSET** Specify 1 or more offsets to add to argument 2. See OFFSET FORMAT below.

**--parse-format=FORMAT, -p FORMAT** Specify a format for parsing DATE-TIME. See PARSE FORMAT below.

**--print-format=FORMAT, --format=FORMAT, -f FORMAT** Specify a format for printing the result. See PRINT FORMAT below.

**--use-task-cycle-time, -c** Use the value of the ROSE\_TASK\_CYCLE\_TIME environment variable as the reference time instead of the current time.

**--utc, -u** Assume date time in UTC instead of local or other time zones.

## CALENDAR MODE

The calendar mode is determined (in order) by:

1. The --calendar=MODE option.
2. The ROSE\_CYCLING\_MODE environment variable.
3. Default to “gregorian”.

## ENVIRONMENT VARIABLES

**ROSE\_CYCLING\_MODE=gregorian|360day|365day|366day** Specify the calendar mode.

**ROSE\_TASK\_CYCLE\_TIME** Specify the current cycle time of a task in a suite. If the --use-task-cycle-time option is set, the value of this environment variable is used by the command as the reference time instead of the current time.

## OFFSET FORMAT

OFFSET must follow the ISO 8601 duration representations such as PnW or PnYnMnDnHnMnS – P followed by a series of nU where U is the unit (Y, M, D, H, M, S) and n is a positive integer, where T delimits the date series from the time series if any time units are used. n may also have a decimal (e.g. PT5.5M) part for a unit provided no smaller units are supplied. It is not necessary to specify zero values for units. If OFFSET is negative, prefix a -. For example:

- P6D - 6 day offset
- PT6H - 6 hour offset
- PT1M - 1 minute offset
- -PT1M - (negative) 1 minute offset
- P3M - 3 month offset
- P2W - 2 week offset (note no other units may be combined with weeks)
- P2DT5.5H - 2 day, 5.5 hour offset
- -P2YT4S - (negative) 2 year, 4 second offset

The following deprecated syntax is supported: OFFSET in the form nU where U is the unit (w for weeks, d for days, h for hours, m for minutes and s for seconds) and n is a positive or negative integer.

## PARSE FORMAT

The format for parsing a date time point should be compatible with the POSIX strftime template format (see the strftime command help), with the following subset supported across all date/time ranges:

%F, %H, %M, %S, %Y, %d, %j, %m, %s, %z

If not specified, the system will attempt to parse DATE-TIME using the following formats:

- ctime: %a %b %d %H:%M:%S %Y
- Unix date: %a %b %d %H:%M:%S %Z %Y
- Basic ISO8601: %Y-%m-%dT%H:%M:%S, %Y%m%dT%H%M%S
- Cylc: %Y%m%d%H

If none of these match, the date time point will be parsed according to the full ISO 8601 date/time standard.

## PRINT FORMAT

For printing a date time point, the print format will default to the same format as the parse format. Also supports the isodatetime library dump syntax for these operations which follows ISO 8601 example syntax - for example:

- CCYY-MM-DDThh:mm:ss -> 1955-11-05T09:28:00,
- CCYY -> 1955,
- CCYY-DDD -> 1955-309,

- CCYY-Www-D -> 1955-W44-6.

Usage of this ISO 8601-like syntax should be as ISO 8601-compliant as possible.

Note that specifying an explicit timezone in this format (e.g. CCYY-MM-DDThh:mm:ss+0100 or CCYYDDDDThhmmZ will automatically adapt the date/time to that timezone i.e. apply the correct hour/minute UTC offset.

For printing a duration, the following can be used in format statements:

- y: years
- m: months
- d: days
- h: hours
- M: minutes
- s: seconds

For example, for a duration P57DT12H - y, m, d, h -> 0, 0, 57, 12

---

### 8.1.10 rose edit

Alias - see [rose config-edit](#) (page 250)

### 8.1.11 rose env-cat

```
rose env-cat [OPTIONS] [FILE ...]
```

Substitute environment variables in input files and print.

If no argument is specified, read from STDIN. One FILE argument may be -, which means read from STDIN.

In match-mode=default, the command will look for \$NAME or \${NAME} syntax and substitute them with the value of the environment variable NAME. A backslash in front of the syntax, e.g. \\$NAME or \\$\${NAME} will escape the substitution.

In match-mode=brace, the command will look for \${NAME} syntax only.

#### OPTIONS

- match-mode=MODE, -m MODE Specify the match mode, which can be brace or default.
  - output=FILE, -o FILE Specify an output file. If no output file is specified or if FILE is -, write output to STDOUT.
  - unbound=STRING, --undef=STRING The command will normally fail on unbound (or undefined) variables. If this option is specified, the command will substitute an unbound variable with the value of STRING, (which can be an empty string), instead of failing.
- 

### 8.1.12 rose host-select

```
rose host-select [OPTIONS] [GROUP/HOST ...]
```

Select a host from a set of groups or names by load, by free memory or by random.

Use settings in \$ROSE\_HOME/etc/rose.conf and \$HOME/.metomi/rose.conf to determine the ranking method.

Print the selected host name.

## OPTIONS

**--choice=N** Choose from any of the top N hosts.

**--debug** Print stack trace on error.

**--quiet, -q** Decrement verbosity.

**--rank-method=METHOD [ :METHOD-ARG ]** Specify the method for ranking a list of hosts. The method can be:

**load** Rank by average load as reported by uptime divided by number of virtual processors.

If METHOD-ARG is specified, it must be 1, 5 or 15. The default is to use the 15 minute load.

**fs** Rank by % usage of a file system as reported by df.

METHOD-ARG must be a valid file system in all the given hosts and host groups. The default is to use the ~ directory.

**mem** Rank by largest amount of free memory. Uses free -m to return memory in Mb

**random** No ranking is used.

**--threshold=[METHOD [ :METHOD-ARG ] : ]VALUE** Each of these option specifies a numeric value of a threshold of which the hosts must either not exceed or must be greater than depending on the specified method . Accepts the same METHOD and METHOD-ARG (and the same defaults) as the --rank-method=METHOD [ :METHOD-ARG ] option. (Obviously, the random method does not make sense in this case.) load and fs must not exceed threshold while mem must be greater than threshold. A host not meeting a threshold condition will be excluded from the ranking list.

**--timeout=FLOAT** Set the timeout in seconds of SSH commands to hosts.

**--verbose, -v** Increment verbosity.

## CONFIGURATION

The command reads its settings from the [rose-host-select] section in \$ROSE\_HOME/etc/rose.conf and \$HOME/.metomi/rose.conf. All settings are optional. Type rose config rose-host-select to print settings. Valid settings are:

**default = GROUP/HOST ...** The default arguments to use for this command.

**group{NAME} = GROUP/HOST ...** Declare a named group of hosts.

**method{NAME} = METHOD [ :METHOD-ARG ]** Declare the default ranking method for a group of hosts.

**thresholds{NAME} = [METHOD [ :METHOD-ARG ] : ]VALUE ...** Declare the default threshold(s) for a group of hosts.

**timeout = FLOAT** Set the timeout in seconds of SSH commands to hosts. (default=10.0)

---

## 8.1.13 rose macro

```
rose macro [OPTIONS] [MACRO_NAME ...]
```

List or run macros associated with a suite or application.

Macros are listed/run according to the config dir (\$PWD unless --config=DIR is set):

- If the config dir is an app directory (or is within an app directory) macros will be listed/run for the `rose-app.conf` file of that app.
- Otherwise macros will be listed/run for the `rose-suite.conf`, `rose-suite.info` and (unless `--suite-only` is set) all `rose-app.conf` files.

If a configuration contains optional configurations:

- For validator macros, validate the main configuration, then validate each main + optional configuration in turn.
- For transform macros, transform the main configuration, then transform each main + optional configuration, recreating each optional configuration as the diff vs the transformed main.

## OPTIONS

**--config=DIR, -C DIR** Use configuration in DIR instead of \$PWD.

**--fix, -F** Prepend all internal transformer (fixer) macros to the argument list.

**--meta-path=PATH, -M PATH** Prepend PATH to the metadata search path (look here first). This option can be used repeatedly to load multiple paths.

**--non-interactive, --yes, -y** Switch off interactive prompting (=answer yes to everything).

**--output=DIR, -O DIR** The location of the output directory. Only meaningful if there is at least one transformer in the argument list.

**--quiet, -q** Reduce verbosity.

**--suite-only** Run only for suite level macros.

**--transform, -T** Prepend all transformer macros to the argument list.

**--validate, -V** Prepend all validator macros to the argument list.

## ARGUMENTS

**MACRO\_NAME ...** A list of macro names to run. If no macro names are specified and `--fix`, `--validate` are not used, list all available macros. Otherwise, run the specified macro names.

## ENVIRONMENT VARIABLES

**optional ROSE\_META\_PATH** Prepend \$ROSE\_META\_PATH to the metadata search path.

### 8.1.14 rose make-docs

```
rose make-docs [OPTIONS] [BUILD...]
```

Build the rose documentation in the requested BUILD format(s).

## OPTIONS

**--venv** Build virtualenv for temporarilly installing python dependencies if necessary.

**--dev** Development mode, don't remove virtualenv after build.

**--strict** Disable cache forcing a complete re-build and turn warnings into errors.

**--debug** Run make with the `-d` option.

**--default-version=VERSION** By default the current version is symlinked as the default version, provide an alternative version to override this.

## BUILD

The format(s) to build the documentation in - default html. Available formats are listed in the sphinx documentation (<http://www.sphinx-doc.org/en/stable/builders.html>). The most commonly used formats are:

**html** For building standalone HTML files.

**singlehtml** For building a single page HTML document.

**latexpdf** For building PDF documentation.

**clean** Removes all built documentation for the current rose version. (use `rm -rf doc` to remove all documentation).

## DEVELOPMENT BUILDS

For development purposes use the following BUILDS:

**doctest** Runs any doctest examples present in documented python modules.

**linkcheck** Checks external links.

---

### 8.1.15 rose metadata-check

```
rose metadata-check [OPTIONS] [ID ...]
```

Validate configuration metadata.

#### OPTIONS

**--config=DIR, -C DIR** The directory containing the configuration metadata i.e. containing the `rose-meta.conf` file, amongst other things. If not specified, the current directory will be used.

**--property=PROPERTY, -p PROPERTY** Only check a certain property e.g. trigger. This can be specified more than once.

#### ARGUMENTS

**ID** One or more sections (configuration IDs) to check in the metadata e.g.`env=FOO` or `namelist:bar`. If specified, only this section will be checked.

---

### 8.1.16 rose metadata-gen

```
rose metadata-gen [OPTIONS] [PROPERTY=VALUE ...]
```

Automatically generate metadata from an application or suite configuration. An aid for metadata development.

**Warning:** May Contain Thorns.

#### OPTIONS

**--auto-type** Add a ‘best guess’ for the `type` and `length` metadata.

**--config=DIR, -C DIR** The directory containing the application or suite configuration to read in. If not specified, the current directory will be used.

**--output=DIR, -O DIR** A directory to output the metadata to. If not specified, output to the application or suite metadata directory.

#### ARGUMENTS

**PROPERTY[=VALUE] ...** One or more key=value pairs of properties to specify for every setting e.g. `compulsory=true`. If `=VALUE` is missing, the property will be set to a null string in each setting.

---

### 8.1.17 rose metadata-graph

```
rose metadata-graph [OPTIONS] [SECTION ...]
```

Graph configuration metadata.

#### OPTIONS

**--config=DIR, -C DIR** The directory containing either the configuration or the configuration metadata. If the configuration is given, the metadata will be looked up in the normal way (see also **--meta-path**, **ROSE\_META\_PATH**). If the configuration metadata is given, there will be no configuration data used in the graphing. If not specified, the current directory will be used.

**--meta-path=PATH, -M PATH** Prepend PATH to the metadata search path (look here first). This option can be used repeatedly to load multiple paths.

**--property=PROPERTY, -p PROPERTY** Graph a certain property e.g. trigger. If specified, only this property will be graphed.

#### ARGUMENTS

**SECTION** One or more configuration sections to graph. If specified, only these sections will be checked.

#### ENVIRONMENT VARIABLES

**optional ROSE\_META\_PATH** Prepend \$ROSE\_META\_PATH to the metadata search path.

### 8.1.18 rose mpi-launch

```
1. rose mpi-launch -f FILE
2. rose mpi-launch
3. rose mpi-launch COMMAND [ARGS ...]
```

Provide a portable way to launch an MPI command.

1. If **--command-file=FILE** (or **-f FILE**) is specified, FILE is assumed to be the command file to be submitted to the MPI launcher command.
2. Alternatively, if \$PWD/rose-mpi-launch.rc exists and **--command-file=FILE** (or **-f FILE**) is not specified, it is assumed to be the command file to be submitted to the MPI launcher command.
3. In the final form, it will attempt to submit COMMAND with the MPI launcher command.

In all cases, the remaining arguments will be appended to the command line of the launcher program.

#### OPTIONS

**--command-file=FILE, -f FILE** Specify a command file for the MPI launcher.

**--debug** Switch on xtrace, i.e. set -x.

**--quiet, -q** Decrement verbosity.

**--verbose, -v** Increment verbosity. Print command on -v mode. Run printenv, ldd on a binary executable, and ulimit -a on -v -v mode.

#### CONFIGURATION

The command reads from the [rose-mpi-launch] section in \$ROSE\_HOME/etc/rose.conf and \$HOME/.metomi/rose.conf. Valid settings are:

**launcher-list=LIST** Specify a list of launcher commands.

E.g:

- launcher-list=poe mpieexec

**launcher-fileopts.LAUNCHER=OPTION-TEMPLATE** Specify the options to a LAUNCHER for launching with a command file. The template string should contain \$ROSE\_COMMAND\_FILE (or \${ROSE\_COMMAND\_FILE}), which will be expanded to the path to the command file.

E.g.:

- launcher-fileopts.mpiexec=-f \$ROSE\_COMMAND\_FILE
- launcher-fileopts.poe=-cmdfile \$ROSE\_COMMAND\_FILE

**launcher-(pre|post)opts.LAUNCHER=OPTION-TEMPLATE** Specify the options to a LAUNCHER for launching with a command. preopts are options placed after the launcher command but before COMMAND. postopts are options placed after COMMAND but before the remaining arguments.

E.g.:

- launcher-preopts.mpiexec=-n \$PROC

## ENVIRONMENT VARIABLES

**optional ROSE\_LAUNCHER** Specify the launcher program.

**optional ROSE\_LAUNCHER\_LIST** Override launcher-list setting in configuration.

**optional ROSE\_LAUNCHER\_FILEOPTS** Override launcher-fileopts.LAUNCHER setting for the selected LAUNCHER.

**optional ROSE\_LAUNCHER\_PREOPTS** Override launcher-preopts.LAUNCHER setting for the selected LAUNCHER.

**optional ROSE\_LAUNCHER\_POSTOPTS** Override launcher-postopts.LAUNCHER setting for the selected LAUNCHER.

**optional ROSE\_LAUNCHER\_ULIMIT\_OPTS** Only relevant when launching with a command. Tell launcher to run rose mpi-launch --inner \$. Specify the arguments to ulimit. E.g. Setting this variable to -a -s unlimited -d unlimited -a results in ulimit -a; ulimit -s unlimited; ulimit -d unlimited; ulimit -a.

**optional NPROC** Specify the number of processors to run on. Default is 1.

## DIAGNOSTICS

Return 0 on success, 1 or exit code of the launcher program on failure.

---

### 8.1.19 rose namelist-dump

```
rose-namelist-dump [OPTIONS] [FILE ...]
```

Convert namelist files into a Rose application configuration snippet. Each argument should be the path to an empty file or a file containing Fortran namelist groups. A – can be used once in the argument list to specify the standard input. If no argument is given, it assumes the standard input is specified. Where possible, use relative path for file names, as the file names appear as-specified in the generated configuration.

#### OPTIONS

**--case=MODE** Output names in lower case (MODE="upper"), upper case (MODE="lower") or unchanged (default).

**--lower, -l** Shorthand for --case=lower.

**--output=FILE, -o FILE** The name of the file for dumping the output. Default is –, i.e. the standard output.

**-u, --upper** Shorthand for --case=upper.

---

## 8.1.20 rose sgc

Alias - see [rose suite-gcontrol](#) (page 261)

## 8.1.21 rose slv

Alias - see [rose suite-log](#) (page 262)

## 8.1.22 rose stem

```
rose stem [options]
```

Run a suitable suite with a specified set of source tree(s).

Default values of some of these settings are suite-dependent, specified in the `rose-suite.conf` file.

### EXAMPLES

```
rose stem --group=developer
rose stem --source=/path/to/source --source=/other/source --group=mygroup
rose stem --source=foo=/path/to/source --source=bar=fcm:bar_tr@head
```

### OPTIONS

All options of [rose suite-run](#) (page 263) are supported. Additional options are:

**--source=SOURCE, -s SOURCE, --source=PROJECT=SOURCE, -s PROJECT=SOURCE**

Specify a source tree to include in a rose-stem suite. The first source tree must be a working copy as the location of the suite and fcm-make config files are taken from it. Further source trees can be added with additional `--source` arguments. The project which is associated with a given source is normally automatically determined using FCM, however the project can be specified by putting the project name as the first part of this argument separated by an equals sign as in the third example above. Defaults to `.` if not specified.

**--group=GROUP [, GROUP2 [, . . . ]], -g GROUP [, GROUP2 [, . . . ]]** Specify a group name to run.

Additional groups can be specified with further `--group` arguments. The suite will then convert the groups into a series of tasks to run..

**--task=TASK [, TASK2 [, . . . ]], -t TASK [, TASK2 [, . . . ]]** Synonym for `--group`.

### JINJA2 VARIABLES

Note that `<project>` refers to the FCM keyword name of the repository in upper case.

**HOST\_SOURCE\_<project>** The complete list of source trees for a given project. Working copies in this list have their hostname prefixed, e.g. `host:/path/wc`.

**HOST\_SOURCE\_<project>\_BASE** The base of the project specified on the command line. This is intended to specify the location of fcm-make config files. Working copies in this list have their hostname prefixed.

**RUN\_NAMES** A list of groups to run in the rose-stem suite.

**SOURCE\_<project>** The complete list of source trees for a given project. Unlike the `HOST_` variable of similar name, paths to working copies do NOT include the host name.

**SOURCE\_<project>\_BASE** The base of the project specified on the command line. This is intended to specify the location of fcm-make config files. Unlike the `HOST_` variable of similar name, paths to working copies do NOT include the host name.

**SOURCE\_<project>\_REV** The revision of the project specified on the command line. This is intended to specify the revision of fcm-make config files.

### 8.1.23 rose suite-clean

```
rose suite-clean [OPTIONS] [SUITE-NAME [...]]
```

Remove items created by “*rose suite-run*”.

If no argument is specified, use the base-name of \$PWD as suite name.

Correctly remove runtime directories created by *rose suite-run* (page 263) including those on remote job hosts.

#### OPTIONS

**--name=NAME, -n NAME** Append NAME to the argument list.

**--non-interactive, --yes, -y** Switch off interactive prompting (=answer yes to everything)

**--only=ITEM** If one or more **--only=ITEM** option is specified, only files and/or directories matching an ITEM will be removed. An ITEM should be a glob pattern (bash extglob) for matching a relative path in the run directory of the suite(s).

**--quiet, -q** Decrement verbosity.

**--verbose, -v** Increment verbosity.

#### DIAGNOSTICS

Return the difference between the number of arguments and number of successfully cleaned suites, i.e. 0 if all successful.

---

### 8.1.24 rose suite-cmp-vc

```
rose suite-cmp-vc NAME
rose suite-cmp-vc --name=NAME

# If CYLC_SUITE_NAME is exported, compare source info of the current
# suite from within a suite task if no name is specified.
rose suite-cmp-vc
```

Compare VCS information of a suite source between installation and now.

Version control system information of a suite is installed under log/rose-suite-run.version file. This command attempts to regenerate the information from the recorded source, and compare the original file with the latest information.

Return 0 if no difference. Print unified diff and return 1 if difference found. Return 2 on other errors.

#### ARGUMENTS

#### OPTIONS

**--name=NAME, -n NAME** Specify the suite NAME.

**--quiet, -q** Decrement verbosity.

**--verbose, -v** Increment verbosity.

---

### 8.1.25 rose suite-gcontrol

```
rose suite-gcontrol [OPTIONS] [--name=SUITE-NAME] [-- EXTRA-ARGS ...]
```

Launch suite engine's suite control GUI for a suite.

If --name=SUITE-NAME is not specified, the name will be determined by locating a `rose-suite.conf` file in `$PWD` or its nearest parent directories. In a normal suite, the basename of the (nearest parent) directory containing the `rose-suite.conf` file is assumed to be the suite name. In a project containing a rose stem suite, the basename of the (nearest parent) directory containing the `rose-stem/rose-suite.conf` file is assumed to be the suite name.

This wrapper is to deal with the use case where a suite may be running on dedicated servers at a site. The wrapper will make an attempt to detect where the suite is running or last run.

#### OPTIONS

- all** Open a suite control GUI for each running suite.
  - host=HOST** Specify a host.
  - name=SUITE-NAME** Specify the suite name.
  - quiet, -q** Decrement verbosity.
  - verbose, -v** Increment verbosity.
- 

### 8.1.26 rose suite-hook

```
# Cylc interface
rose suite-hook [OPTIONS] EVENT SUITE MSG
rose task-hook [OPTIONS] EVENT SUITE TASK_ID MSG
```

Deprecated. Use cylc built-in event handlers instead.

Provide a common event hook for cylc suites and tasks.

- (Task event only) Retrieve remote task logs back to server host.
- Email user if --mail specified.
- Shutdown suite if --shutdown specified.

#### OPTIONS

- debug** Switch on debug mode.
  - mail-cc=LIST** Only useful if the --mail option is specified. Specify a comma separated list of additional addresses to email.
  - mail** Trigger an email notification to the user.
  - retrieve-job-logs** Retrieve remote task job logs.
  - shutdown** Trigger a shutdown of the suite.
  - quiet, -q** Decrement verbosity.
  - verbose, -v** Increment verbosity.
- 

### 8.1.27 rose suite-init

Alias - see [rose suite-run](#) (page 263)

### 8.1.28 rose suite-log

```
1. rose suite-log [--view]
2. rose suite-log --update [ITEM ...]
   rose suite-log --update '*' # all task jobs
3. rose suite-log --archive CYCLE ...
   rose suite-log --archive '*' # all cycles
```

View or update suite log.

1. Launch web browser to view suite log. If “rose bush” is not configured, the command will offer to start it.
2. Pull back task job logs from any remote hosts for specified cycle times or task names or IDs.
3. Archive (tar-gzip) job logs at the specified cycle time.

If `--name=SUITE-NAME` is not specified, the name will be determined by locating a `rose-suite.conf` file in `$PWD` or its nearest parent directories. In a normal suite, the basename of the (nearest parent) directory containing the `rose-suite.conf` file is assumed to be the suite name. In a project containing a rose stem suite, the basename of the (nearest parent) directory containing the `rose-stem/rose-suite.conf` file is assumed to be the suite name.

#### OPTIONS

**--archive** Archive (tar-gzip) job logs at specified cycle times. Implies `--update`.

**--force, -f** Same as `rose suite-log --update '*'.`

**--name=SUITE-NAME, -n SUITE-NAME** Specify the suite name, instead of using basename of `$PWD`.

**--prune-remote** If specified, remove job logs from remote hosts after pulling them to suite host.

**--tidy-remote** Deprecated. Use `--prune-remote` instead.

**--update, -U** Update job logs for items specified in arguments.

**--user=USER-NAME, -u USER-NAME** View mode only. View logs of a suite of a different user.

**--view** Launch web browser to view suite log.

---

### 8.1.29 rose suite-log-view

Alias - see *rose suite-log* (page 262)

### 8.1.30 rose suite-restart

```
rose suite-restart [OPTIONS] [[--] CYLC-RESTART-ARGS]

# Restart cylc suite with name equal to basename of $PWD.
rose suite-restart

# Restart cylc suite NAME
rose suite-restart --name=NAME

# Restart cylc suite NAME with a given state dump
rose suite-restart --name=NAME -- state.20141118T161121.195326Z

# Restart cylc suite NAME on given host
rose suite-restart --name=NAME --host=my-suite-host
```

Restart a shutdown suite from its last known state without reinstalling it.

If re-installation is required, use `rose suite-run --restart`.

## ARGUMENTS

### OPTIONS

**--host=HOST** Specify a host for restarting the suite.

**--name=NAME, -n NAME** Specify the suite NAME in cylc, instead of using the basename of the current working directory.

**--no-gcontrol** Do not run `cylc gui`. Default is to run `cylc gui` if the suite is running and the `DISPLAY` environment variable is defined.

**--quiet, -q** Decrement verbosity.

**--verbose, -v** Increment verbosity.

### 8.1.31 rose suite-run

```
rose suite-run [OPTIONS] [[--] CYLC-RUN-ARGS]

# Install and run a Cylc suite in $PWD.
rose suite-run

# As above, but start the suite in simulation mode.
rose suite-run -- --mode=simulation

# Install and run the suite in $PWD, and register it as "my.suite".
rose suite-run -n my.suite

# Install and run suite in "/dir/to/my.suite".
# Equivalent to (cd /dir/to/my.suite && rose suite-run).
rose suite-run -C /dir/to/my.suite
```

Install and run a Cylc suite.

Install a suite (in \$PWD), register it in Cylc using the basename of the configuration directory or the option specified in `--name=NAME`. Invoke `cylc run` on it. Arguments (and options after `--`) are passed to `cylc run`.

### OPTIONS

**--config=DIR, -C DIR** Specify the configuration directory of the suite. (default=\$PWD)

**--define=[SECTION]KEY=VALUE, -D [SECTION]KEY=VALUE** Each of these overrides the [SECTION]KEY setting with a given VALUE. Can be used to disable a setting using the syntax `--define=[SECTION]!KEY` or even `--define=[!SECTION]`. See also `--define-suite`.

**--define-suite=KEY=VALUE, -S KEY=VALUE** As `--define`, but with an implicit [SECTION] for suite variables.

**--host=HOST** Specify a host for running the suite.

**--install-only, -i** Install the suite. Do not run it. Implies `--no-gcontrol`.

**--local-install-only, -l** Install the suite locally. Do not install to job hosts. Do not run it. Implies `--no-gcontrol`.

**--validate-suite-only** Install the suite in a temporary location. Do not setup any directories. Do not run any jobs. Implies `--no-gcontrol`. Overrides `--install-only` and `--local-install-only`.

**--log-keep= DAYS** Specify the number of days to keep log directories/archives. Do not housekeep if not specified. Named log directories (created by `--log-name=NAME` in previous runs) will not be housekept.

**--log-name=NAME** Specify a name for the log directory of the current run. If specified, it will create a symbolic link `log.NAME` to point to the log directory of the current run. Named log directories will not be automatically archived or housekept. Only works with `--run=run`.

**--name=NAME, -n NAME** Specify the suite NAME in Cylc, instead of using the basename of the configuration directory.

**--new, -N** (Re-)create suite runtime locations. This option is equivalent to running `rose suite-clean -y && rose suite-run` and will remove previous runs. Users may want to take extra care when using this option.

**--no-gcontrol** Do not run `cylc gui`. Default is to run `cylc gui` if the suite is running and the `DISPLAY` environment variable is defined.

**--no-log-archive** Do not archive (tar-gzip) old log directories.

**--no-overwrite** Do not overwrite existing files.

**--no-strict** Do not validate (suite engine configuration) in strict mode.

**--opt-conf-key=KEY, -O KEY, --opt-conf-key='(KEY)', -O '(KEY)'** Each of these switches on an optional configuration identified by KEY. The configurations are applied first-to-last. The (KEY) syntax denotes an optional configuration that can be missing. Otherwise, the optional configuration must exist.

**--quiet, -q** Decrement verbosity.

**--reload** Shorthand for `--run=reload`.

**--restart** Shorthand for `--run=restart`.

**--run=reload|restart|run** Invoke `cylc reload|restart|run` according to this option. (default=`run`)

**--verbose, -v** Increment verbosity.

## ENVIRONMENT VARIABLES

`rose suite-run` (page 263) provides the following environment variables to the suite.

**ROSE\_ORIG\_HOST** The name of the host where the `rose suite-run` (page 263) command was invoked.

**optional ROSE\_SUITE\_OPT\_CONF\_KEYS** Each KEY in this space delimited list switches on an optional configuration. The configurations are applied first-to-last.

## JINJA2 VARIABLES

`rose suite-run` (page 263) provides the following Jinja2 variables to the suite.

**ROSE\_ORIG\_HOST** The name of the host where the `rose suite-run` (page 263) command was invoked.

## SEE ALSO

- `cylc help gui`
  - `cylc help register`
  - `cylc help run`
- 

### 8.1.32 `rose suite-scan`

```
rose suite-scan [HOST ...]
```

Scan for running suites.

If no HOST specified, use `[rose-suite-run] hosts` setting in site / user configuration. If no configuration defined, use localhost.

## OPTIONS

**--quiet, -q** Decrement verbosity.  
**--verbose, -v** Increment verbosity.

---

### 8.1.33 rose suite-shutdown

```
rose suite-shutdown [OPTIONS] [--name=SUITE-NAME] [[--] EXTRA-ARGS ...]
```

Shutdown a running suite.

If --name=SUITE-NAME is not specified, the name will be determined by locating a `rose-suite.conf` file in `$PWD` or its nearest parent directories. In a normal suite, the basename of the (nearest parent) directory containing the `rose-suite.conf` file is assumed to be the suite name. In a project containing a rose stem suite, the basename of the (nearest parent) directory containing the `rose-stem/rose-suite.conf` file is assumed to be the suite name.

This wrapper also deals with the use case where a suite may be running on dedicated servers at a site. The wrapper will make an attempt to detect where the suite is running or last run.

## OPTIONS

**--all** Shutdown all running suites. You will be prompted to confirm shutting down each affected suite.  
**--host=HOST** Specify a host.  
**--name=SUITE-NAME** Specify the suite name.  
**--non-interactive, --yes, -y** Switch off interactive prompting (=answer yes to everything)  
**--quiet, -q** Decrement verbosity.  
**--verbose, -v** Increment verbosity.  
**-- --EXTRA-ARGS** See the `cyclc` documentation, `cyclc shutdown` for options and details on EXTRA-ARGS.

---

### 8.1.34 rose suite-stop

Alias - see `rose suite-shutdown` (page 265)

### 8.1.35 rose task-env

```
rose task-env [OPTIONS]
eval $(rose task-env)
```

Provide an environment for cycling suite task.

Print KEY=VALUE of the following to the STDOUT:

**ROSE\_SUITE\_DIR** The path to the root directory of the running suite.

**ROSE\_SUITE\_DIR\_REL** The path to the root directory of the running suite relative to `$HOME`.

**ROSE\_SUITE\_NAME** The name of the running suite.

**ROSE\_TASK\_NAME** The name of the suite task.

**ROSE\_TASK\_CYCLE\_TIME** The cycle time of the suite task, if there is one.

**ROSE\_CYCLING\_MODE** The cycling mode of the running suite.

**ROSE\_TASK\_LOG\_ROOT** The root path for log files of the suite task.

**ROSE\_DATA** The path to the data directory of the running suite.

**ROSE\_DATAC** The path to the data directory of this cycle time in the running suite.

**ROSE\_DATAC????** The path to the data directory of the cycle time with an offset relative to the current cycle time. ???? is a duration:

A \_\_ (double underscore) prefix denotes a cycle time in the future (because a minus sign cannot be used in an environment variable). Otherwise, it is a cycle time in the past.

The rest should be either an ISO 8601 duration, such as:

- P2W - 2 weeks
- PT12H - 12 hours
- P1DT6H - 1 day, 6 hours
- P4M - 4 months
- PT5M - 5 minutes

Or, for the case of integer cycling suites:

- P1 - 1 cycle before the current cycle
- P5 - 5 cycles before the current cycle

Deprecated syntax:

- nW denotes n weeks.
- n or nD denotes n days.
- Tn or TnH denotes n hours.
- TnM denotes n minutes.
- TnS denotes s seconds.

E.g. ROSE\_DATACPT6H is the data directory of 6 hours before the current cycle time.

E.g. ROSE\_DATACP1D and ROSE\_DATACPT24H are both the data directory of 1 day before the current cycle time.

**ROSE\_ETC** The path to the etc directory of the running suite.

**ROSE\_TASK\_PREFIX** The prefix in the task name.

**ROSE\_TASK\_SUFFIX** The suffix in the task name.

## OPTIONS

**--cycle=TIME, -t TIME** Specify the cycle time. If not defined, use the cycle time provided by the suite environment. TIME can be in an ISO date/time format, CCYYMMDDhh (deprecated) date/time format, or a TIME-DELTA string described in the --cycle-offset=TIME-DELTA option.

**--cycle-offset=TIME-DELTA, -T TIME-DELTA** Specify one or more cycle offsets to determine what ROSE\_DATAC???? environment variables to export. The TIME-DELTA argument uses the syntax explained above in the ROSE\_DATAC???? environment variable.

E.g. --cycle-offset=PT3H --cycle-offset=PT6H will tell *rose task-env* (page 265) to export ROSE\_DATACPT3H and ROSE\_DATACPT6H.

---

**Note:** The main usage of this option is to reference a cycle time in the past, so a positive offset is used to go backward in time, and a negative offset is used to go forward in time.

---

E.g. `--cycle-offset=-PT3H` will tell *rose task-env* (page 265) to export ROSE\_DATA\_C\_\_PT3H for ROSE\_DATA\_C of 3 hours ahead of the current cycle time.

**--path=[NAME=]PATTERN, -P [NAME=]PATTERN** Each of these specify a glob pattern for paths to prepend to an environment variable called NAME (or PATH if NAME is not specified). If a relative path is given, it is relative to \$ROSE\_SUITE\_DIR. An empty value resets the default and any previous --path=PATTERN settings. (Default for PATH is "share/fcm[\_-]make\*/\*/bin" and "work/fcm[\_-]make\*/\*/bin")

**--prefix-delim=DELIMITER** Specify the delimiter used to determine the task name prefix. (Default=\_)

**--suffix-delim=DELIMITER** Specify the delimiter used to determine the task name suffix. (Default=\_)

**--quiet, -q** Decrement verbosity.

**--verbose, -v** Increment verbosity.

## USAGE IN SUITES

*rose task-env* can be used to make environment variables available to a suite by defining its `suite.rc` env-script option as `env-script = eval $(rose task-env)`.

### 8.1.36 rose task-hook

Alias - see *rose suite-hook* (page 261)

### 8.1.37 rose task-run

```
rose task-run [OPTIONS] [--] [APP-COMMAND ...]
```

Provide an environment to run a suite task.

Provides environment variables documented in *rose task-env* (page 265). It is worth noting that if the environment variables are already provided by *rose task-env* (page 265), this command will not override them.

Normally, the suite task will select a Rose application configuration that has the same name as the task. This can be overridden by the `--app-key=KEY` option or the `ROSE_TASK_APP` environment variable.

#### OPTIONS

All options of *rose app-run* (page 245) and *rose task-env* (page 265) are supported. Additional options are:

**--app-key=KEY** Specify a named application configuration.

#### ENVIRONMENT VARIABLES

All environment variables of *rose app-run* (page 245) and *rose task-env* (page 265) are supported. All environment variables documented in *rose task-env* (page 265) are passed to the application *rose task-run* (page 267) runs. The following environment variables are used by *rose task-run* (page 267):

**ROSE\_TASK\_APP** Specify a named application configuration.

#### SEE ALSO

- *rose app-run* (page 245)
- *rose task-env* (page 265)

### 8.1.38 rose test-battery

```
rose test-battery
```

Run Rose self tests.

Change directory to Rose source tree, and runs this shell command:

```
exec prove -j "$NPROC" -s -r "${@:-t}"
```

where NPROC is the number of processors on your computer (or the setting [t]prove-options in the site/user configuration file). If you do not want to run the full test suite, you can specify the names of individual test files or their containing directories as extra arguments.

## EXAMPLES

```
# Run the full test suite with the default options.  
rose test-battery  
# Run the full test suite with 12 processes.  
rose test-battery -j 12  
# Run only tests under "t/rose-app-run/" with 12 processes.  
rose test-battery -j 12 t/rose-app-run  
# Run only "t/rose-app-run/07-opt.t" in verbose mode.  
rose test-battery -v t/rose-app-run/07-opt.t
```

## SEE ALSO

- [prove\(1\)](#)
- 

## 8.1.39 rose tutorial

```
# Print list of available suites.  
rose tutorial  
# Copy SUITE to DIR (defaults to ~/cylc-run/SUITE) .  
rose tutorial SUITE [DIR]
```

Make a copy of one of the tutorial SUITE in the cylc-run directory.

## 8.2 Rosie Commands

### 8.2.1 rosie checkout

```
rosie checkout [OPTIONS] ID ...
```

Checkout local copies of suites.

For each ID in the argument list, checkout a working copy of the suite identified by ID to the standard location.

#### OPTIONS

- force, -f If working copy for suite identified by ID already exists, remove it. Continue to the next ID if checkout of a suite fails.
  - quiet, -q Decrement verbosity.
  - verbose, -v Increment verbosity.
-

## 8.2.2 rosie co

Alias - see [rosie checkout](#) (page 268)

## 8.2.3 rosie copy

Alias - see [rosie create](#) (page 269)

## 8.2.4 rosie create

```
rosie create [OPTIONS]
rosie copy [OPTIONS] FROM-ID
```

Create a new suite, and optionally copy items from an existing one.

Assign an ID and create the directory structure in the central repository for a new suite.

The location of the repository for the new suite is determined in order of preference:

1. --prefix=PREFIX option
2. prefix of the FROM-ID
3. [rosie-id]prefix-default option in the site/user configuration.

If FROM-ID is specified, copy items from the existing suite FROM-ID when the suite is created. It is worth noting that revision history of the copied items can only be preserved if FROM-ID is in the same repository of the new suite.

The syntax of the FROM-ID is PREFIX-xxNNN[/BRANCH][@REV] (e.g. my-su173, my-su173/trunk, [my-su173/trunk@HEAD](#)). If REV is not specified, the last changed revision of the branch is used. If BRANCH is not specified, “trunk” is used.

### OPTIONS

- info-file=FILE** Specify a FILE containing the discovery information for the new suite. If FILE is -, read from STDIN. The default behaviour is to open an editor to add suite discovery information.
- meta-suite** (Admin-only) Create the special suite in the repository containing discovery metadata and known keys.
- no-checkout** Do not checkout a working copy of the newly created suite. Default is to checkout.
- non-interactive, --yes, -y** Switch off interactive prompting (=answer yes to everything)
- prefix=PREFIX** Specify the prefix (i.e. the suite repository) to use.
- project=PROJECT** Specify a project to check/query any available metadata. The default behaviour is to use no project and metadata.
- quiet, -q** Decrement verbosity.
- verbose, -v** Increment verbosity.

---

## 8.2.5 rosie delete

```
rosie delete [OPTIONS] [--] [ID ...]
```

Delete suites.

Check the standard working copy location for a checked out suite matching ID and remove it if there is no uncommitted change (or if **--force** is specified).

Delete the suite directory structure from the HEAD of the central repository matching the ID.

If no ID is specified and \$PWD is a working copy of a suite, use the ID of the suite in the working copy.

#### OPTIONS

**--force, -f** Remove working copies even if there are uncommitted changes. Continue with the next ID if delete of a suite fails.

**--local-only** Remove only the working copy of a suite.

**--non-interactive, --yes, -y** Switch off interactive prompting (=answer yes to everything)

**--quiet, -q** Decrement verbosity.

**--verbose, -v** Increment verbosity.

---

### 8.2.6 rosie disco

```
rosie disco start [PORT] # start ad-hoc web service server (on PORT)
rosie disco stop          # stop ad-hoc web service server
rosie disco stop -y       # stop ad-hoc web service server w/o prompting
rosie disco              # print status of ad-hoc web service server
```

Start/stop ad-hoc Rosie suite discovery web service server.

For `rosie disco start`, if PORT is not specified, use port 8080.

#### OPTIONS

**--non-interactive, --yes, -y** (For stop only.) Switch off interactive prompting (=answer yes to everything)

**--service-root, -R** (For start only.) Include web service name under root of URL.

---

### 8.2.7 rosie go

```
rosie go [OPTIONS] [...]
```

Launch the Rosie client GTK+ GUI.

If arguments are specified, `rosie go` will perform an initial lookup based on the arguments, which can be an address, a query, or search words. See [rosie lookup](#) (page 273) for detail.

Unless an option is used to specify the initial search type the argument is interpreted as follows:

- A string beginning with “http”: an address
- A string not beginning with “http”: search words

An address URL may contain shell meta characters, so remember to put it in quotes.

Search strings may contain SQL wildcard characters. E.g.

- % (percent) is a substitute for zero or more characters.
- \_ (underscore) is a substitute for a single character.

If no argument is specified, `rosie go` will display a list of all your locally checked out suites.

## OPTIONS

- address-mode, --url, -A, -U** Shorthand for `--lookup-mode=address`.
- all-revs** Specify whether to search deleted suites and superceded suites.
- lookup-mode=MODE** Specify the initial lookup mode. MODE can be address, query or search.
- prefix=PREFIX** Specify the name of a Rosie web service to use. This option can be used multiple times.
- query, -Q** Shorthand for `--lookup-mode=query`.
- search, -S** Shorthand for `--lookup-mode=search`.

## SEE ALSO

- [rosie lookup](#) (page 273)
- [rosie ls](#) (page 273)

## 8.2.8 rosie graph

```
rosie graph [OPTIONS] [ID]
```

Graph suite copy ancestry.

## OPTIONS

- distance=DISTANCE, -d DISTANCE** The maximum distance (graph depth) for suites related to ID to be plotted. For example, if the distance is 1, only the parents and children (but not siblings) of ID will be plotted. If not given, this is unlimited. Requires ID to be specified.
- output=FILE, -o FILE** The name of the file for dumping the output. Otherwise, the output will go to a temporary file which will get tidied up. The extension of the filename determines the output format - see graphviz AGraph.draw documentation.
- prefix=PREFIX** Display suites from only the specified web services. This option can be used multiple times.
- text** Print parent and child suites of a suite ID. For example, for a suite “bar” you may get results like:

- [parent] foo
- [child1] baz
- [child1] qux
- [child2] quux
- [child3] corge

where “foo” is the parent of “bar”, “baz” and “qux” its first generation children, “quux” its second generation child and “corge” its third generation child. Also supports use of the `--property` option for producing output. Requires ID to be specified.

- property=PROPERTY, -p PROPERTY** Add a certain suite property to the node labels - e.g. owner or title. This option can be used multiple times.

## ARGUMENTS

- ID** A suite id to graph. If given, only the suites that are connected to this id by copy history will be graphed.

## 8.2.9 rosie hello

```
rosie hello [--prefix=PREFIX]
```

Set up connection to one or more Rosie web service servers.

### OPTIONS

**--prefix=PREFIX** Specify the name of one or more Rosie web service servers to use. This option can be used multiple times.

---

## 8.2.10 rosie id

```
rosie id [OPTIONS]

# Print the repository URL of a given suite ID
rosie id --to-origin mol-abc45

# Print the local location of a given suite ID
rosie id --to-local-copy mol-abc45

# Print the output directory of a given suite ID
rosie id --to-output mol-abc45

# Print the web URL of a given suite ID
rosie id --to-web mol-abc45

# Print suite ID of working copy in $PWD
rosie id

# Print suite ID of working copy in a directory
rosie id /path/to/working/copy

# Print suite ID of a given URL
rosie id svn://fcm1/rose_mol_svn/a/b/c/4/5

# Print latest suite ID in the default repository
rosie id --latest

# Print latest suite ID in the given repository
rosie id --latest mot

# Print next suite ID in the default repository
rosie id --next
```

Utility for working with suite IDs.

### OPTIONS

**--latest** Print the latest suite ID in the repository  
**--to-local-copy** Print the local location of a given suite ID  
**--to-origin** Print the repository URL of a given suite ID  
**--to-output** Print the output directory of a given suite ID  
**--to-web** Print the web URL of a given suite ID  
**--next** Print the next available suite ID in the repository

---

## 8.2.11 rosie lookup

```
rosie lookup [OPTIONS] LOOKUP-TEXT ...
```

Find suites in the suite discovery database.

Search for suites using an address, a query or search words and display the information of the matching suites.

Unless an option is used to specify the initial search type the argument is interpreted as follows:

- A string beginning with “http”: an address
- A string not beginning with “http”: search words

An address URL may contain shell meta characters, so remember to put it in quotes.

The default output format includes a local working copy status field (%local) in the first column.

- A blank field means there is no related suite checked out.
- = means that the suite is checked out at this branch and revision.
- < means that the suite is checked out but at an older revision.
- > means that the suite is checked out but at a newer revision.
- S means that the suite is checked out but on a different branch.
- M means that the suite is checked out and modified.
- X means that the suite is checked out but is corrupted.

Search strings may contain SQL wildcard characters. E.g:

- % (percent) is a substitute for zero or more characters.
- \_ (underscore) is a substitute for a single character.

### OPTIONS

```
--address-mode, --url, -A, -U Shorthand for --lookup-mode=address.
--all-revs Specify whether to search deleted suites and superceded suites.
--no-headers, -H Do not print column headers.
--lookup-mode=MODE Specify the initial lookup mode. MODE can be address, query or search.
--prefix=PREFIX Specify the name of a Rosie web service to use. This option can be used multiple times.
--print-format=FORMAT, --format=FORMAT, -f FORMAT Control the output format of the results
    using a string containing column names or properties preceded by %. For example: rosie lookup
    daisy --format="%idx from %owner" might give: abc01 from daisy
--query, -Q Shorthand for --lookup-mode=query.
--quiet, -q Shorthand for --format="%idx.
--reverse, -r Reverse sort order.
--search, -S Shorthand for --lookup-mode=search.
--sort=FIELD, -s FIELD Sort results by the field FIELD instead of revision.
--verbose, -v Display full info for each returned suite.
```

---

## 8.2.12 rosie ls

```
rosie ls [OPTIONS]
```

List the local suites.

Search for locally checked out suites and print their details.

The default format includes a local working copy status field (%local) in the first column. A blank field means there is no related suite checked out.

- = means that the suite is checked out at this branch and revision.
- < means that the suite is checked out but at an older revision.
- > means that the suite is checked out but at a newer revision.
- S means that the suite is checked out but on a different branch.
- M means that the suite is checked out and modified.
- X means that the suite is checked out but is corrupted.

## OPTIONS

**--prefix=PREFIX** Display locally checked out suites from only the specified web services. This option can be used multiple times.

**--no-headers, -H** Do not print column headers.

**--print-format=FORMAT, --format=FORMAT, -f FORMAT** Control the output format of the results using a string containing column names or properties preceded by %. For example: rosie ls --format="%idx from %owner" might give: abc01 from daisy

**--quiet, -q** Shorthand for --format="%idx".

**--reverse, -r** Reverse sort order.

**--sort=FIELD, -s FIELD** Sort results by the field FIELD instead of revision.

**--user=~USERNAME, -u ~USERNAME** Specify another user whose roses directory you want to list e.g.  
--user=~bob

**--verbose, -v** Display full info for each returned suite.

## ROSE ENVIRONMENT VARIABLES

### NPROC

**Description** Specifies the number of processors to run on. Default is 1.

#### Used By

- *rose mpi-launch* (page 257)

### ROSE\_APP\_COMMAND\_KEY

**Description** Can be set to define which command in an app config to use.

#### Used By

- *rose app-run* (page 245)
- *rose task-run* (page 267)

### ROSE\_APP\_OPT\_CONF\_KEYS

**Description** Each KEY in this space delimited list switches on an optional configuration in an application. The (KEY) syntax can be used to denote an optional configuration that can be missing. The configurations are applied in first-to-last order.

#### Used By

- *rose app-run* (page 245)
- *rose task-run* (page 267)

### ROSE\_BUNCH\_LOG\_PREFIX

**Description** Environment variable provided to Rose Bunch instances at runtime to identify the log prefix that will be used for output e.g. for a bunch instance named `foo` then `ROSE_BUNCH_LOG_PREFIX=foo`.

#### Provided At Runtime By

- Rose Bunch

### ROSE\_CONF\_PATH

**Description** Specify a colon (:) separated list of paths for searching and loading site/user configuration. If this environment variable is not defined, the normal behaviour is to search for and load *rose.conf* (page 198) from `$ROSE_HOME/etc` and then `$HOME/.metomi`.

#### Used By

- *rose test-battery* (page 267)

### ROSE\_CYCLING\_MODE

**Description** The cycling mode to use when manipulating dates. Can be either `360day` or `gregorian`.

#### Used By

- *rose date* (page 250)

**Provided By**

- [rose task-env](#) (page 265)

**ROSE\_DATA**

**Description** The path to the data directory of the running suite.

**Provided By**

- [rose task-env](#) (page 265)

**ROSE\_DATAC**

**Description** The path to the data directory of this cycle time in the running suite.

**Provided By**

- [rose task-env](#) (page 265)

**ROSE\_DATAC????**

**Description** The path to the data directory of the cycle time with an offset relative to the current cycle time.  
???? is a duration:

- A \_\_ (double underscore) prefix denotes a cycle time in the future. Otherwise, it is a cycle time in the past.
- PnM denotes  $n$  months.
- PnW denotes  $n$  weeks.
- PnD or nD denotes  $n$  days.
- PTnH or TnH denotes  $n$  hours.
- PTnM denotes  $n$  minutes.

E.g. ROSE\_DATACPT6H is the data directory of 6 hours before the current cycle time.

E.g. ROSE\_DATACP1D and ROSE\_DATACPT24H are both the data directory of 1 day before the current cycle time.

**Provided By**

- [rose task-env](#) (page 265)

**ROSE\_ETC**

**Description** The path to the etc directory of the running suite.

**Provided By**

- [rose task-env](#) (page 265)

**ROSE\_FILE\_INSTALL\_ROOT**

**Description** If specified, change to the specified directory to install files.

**Used By**

- [rose app-run](#) (page 245)
- [rose task-run](#) (page 267)

**ROSE\_HOME**

**Description** Specifies the path to the Rose home directory.

**Used and Provided By**

- rose

**ROSE\_HOME\_BIN**

**Description** Specifies the path to the bin/ or sbin/ directory of the current Rose utility.

**Used and Provided By**

- *rose*

**ROSE\_LAUNCHER**

**Description** Specifies the launcher program to run the prog.

**Used By**

- *rose mpi-launch* (page 257)

**ROSE\_LAUNCHER\_FILEOPTS**

**Description** Override [rose-mpi-launch]launcher-fileopts.LAUNCHER setting for the selected *ROSE\_LAUNCHER* (page 277).

**Used By**

- *rose mpi-launch* (page 257)

**ROSE\_LAUNCHER\_LIST**

**Description** Specifies an alternative list of launchers.

**Used By**

- *rose mpi-launch* (page 257)

**ROSE\_LAUNCHER\_PREOPTS**

**Description** Override [rose-mpi-launch]launcher-preopts.LAUNCHER setting for the selected *ROSE\_LAUNCHER* (page 277).

**Used By**

- *rose mpi-launch* (page 257)

**ROSE\_LAUNCHER\_POSTOPTS**

**Description** Override [rose-mpi-launch]launcher-postopts.LAUNCHER setting for the selected *ROSE\_LAUNCHER* (page 277).

**Used By**

- *rose mpi-launch* (page 257)

**ROSE\_LAUNCHER\_ULIMIT\_OPTS**

**Description** Tell launcher to run:

```
rose mpi-launch --inner $@
```

Specify the arguments to ulimit. E.g. Setting this variable to:

```
-a -s unlimited -d unlimited -a
```

results in:

```
ulimit -a; ulimit -s unlimited; ulimit -d unlimited; ulimit -a
```

**Used By**

- *rose mpi-launch* (page 257)

**ROSE\_META\_PATH**

**Description** Defines a metadata search path, colon-separated for multiple paths.

**Used by**

- *rose config-edit* (page 250)

- *rose macro* (page 254)

#### **ROSE\_NS**

**Description** Defines the rose namespace. Used to identify if a utility belongs to `rose` or `rosie`.

**Used and Provided By**

- `rose`

#### **ROSE\_ORIG\_HOST**

**Description** The name of the host where the *rose suite-run* (page 263) command was invoked.

**Provided By**

- *rose suite-run* (page 263)

#### **ROSE\_SUITE\_DIR**

**Description** The path to the root directory of the running suite.

**Provided By**

- *rose task-env* (page 265)

#### **ROSE\_SUITE\_DIR\_REL**

**Description** The path to the root directory of the running suite relative to `$HOME`.

**Provided By**

- *rose task-env* (page 265)

#### **ROSE\_SUITE\_NAME**

**Description** The name of the running suite.

**Provided By**

- *rose task-env* (page 265)

#### **ROSE\_SUITE\_OPT\_CONF\_KEYS**

**Description** Each KEY in this space delimited list switches on an optional configuration when installing a suite. The (KEY) syntax can be used to denote an optional configuration that can be missing. The configurations are applied in first-to-last order.

**Used By**

- *rose suite-run* (page 263)

#### **ROSE\_TASK\_APP**

**Description** Specify a named application configuration.

**Used By**

- *rose task-run* (page 267)

#### **ROSE\_TASK\_CYCLE\_TIME**

**Description** The cycle time of the suite task, if there is one.

**Provided By**

- *rose task-env* (page 265)

#### **ROSE\_TASK\_LOG\_DIR**

**Description** The directory for log files of the suite task.

**Provided By**

- *rose task-env* (page 265)

#### **ROSE\_TASK\_LOG\_ROOT**

**Description** The root path for log files of the suite task.

**Provided By**

- [rose task-env](#) (page 265)

#### ROSE\_TASK\_N\_JOBS

**Warning:** (Deprecated) Use the `opt.jobs` setting in the application configuration instead.

**Description** The number of jobs to run in parallel in `fcm make` (`default=4`).

**Used By**

- `fcm_make` built-in application
- `fcm_make2` built-in application

#### ROSE\_TASK\_MIRROR\_TARGET

**Warning:** (Deprecated)

**Description** The mirror target for the mirror step in the `fcm-make.cfg` configuration.

**Provided By**

- `fcm_make` built-in application

#### ROSE\_TASK\_NAME

**Description** The name of the suite task.

**Provided By**

- [rose task-env](#) (page 265)

**Used By**

- [rose app-run](#) (page 245)

#### ROSE\_TASK\_OPTIONS

**Warning:** (Deprecated) Use the `args` setting in the application configuration instead.

**Description** Additional options and arguments for `fcm make` or [rose app-run](#) (page 245).

**Used By**

- `fcm_make` built-in application
- `fcm_make2` built-in application

#### ROSE\_TASK\_PREFIX

**Description** The prefix in the task name.

**Provided By**

- [rose task-env](#) (page 265)

#### ROSE\_TASK\_SUFFIX

**Description** The suffix in the task name.

**Provided By**

- *rose task-env* (page 265)

**ROSE\_UTIL**

**Description** Used to identify which `rose` or `rosie` utility is being run.

**Used and Provided By**

- `rose`

**ROSE\_VERSION**

**Description** The current version of Rose.

**Used and Provided By**

- `rose`
- *rose suite-run* (page 263)

## ROSE BASH LIBRARY

The Rose bash library lives in `lib/bash/`. To import a module, load the file into your script. E.g. To load `rose_usage`, you would do:

```
 . $ROSE_HOME/lib/bash/rose_usage
```

The modules are:

**`rose_init`** Called by `rose` on initialisation. This is not meant to be for general use.

**`rose_log`** Provide functions to print log messages.

**`rose_usage`** If your script has a header similar to the ones used by a `rose` command line utility, you can use this function to print the synopsis section of the script header.



## ROSE BUILT-IN APPLICATIONS

Rose contains a few built-in applications providing common functionality.

These applications can be run using [rose task-run](#) (page 267) or [rose app-run](#) (page 245).

To use a built-in application, add the `rose-app.conf/mode=KEY` (page 194) setting in the application configuration, where KEY is the name of the built-in application. For example to use the [rose\\_prune](#) (page 293) built-in application:

Listing 1: rose-app.conf

```
mode=rose_prune
# There may also be metadata which can be picked up using the `meta` setting.
meta=rose_prune
```

A built-in application would normally behave very much like running an external command. The key differences are normally that:

- A built-in application may use a different working directory to a Rose application.
- A built-in application will run some pre-defined commands or logic instead of a command defined in the application configuration.

### 11.1 fcm\_make

An application for running the `fcm make` command.

The `fcm_make` application expects a file `file/fcm-make.cfg` in its application configuration. It runs `fcm make` using this configuration file.

You can configure these applications with environment variables or settings in `rose-app.conf`. (Settings in `rose-app.conf` override their equivalent environment variables.)

By default the `bin/` directories of builds will be prepended to the `PATH` environment variable by [rose task-env](#) (page 265) and/or [rose task-run](#) (page 267) commands run by subsequent tasks in the suite.

#### 11.1.1 Example

```
meta=fcm_make
mode=fcm_make
opt.jobs=8
```

#### 11.1.2 Invocation

- If a task's name contains the string `fcm_make` then [rose task-run](#) (page 267) will run this built-in application automatically.

- If a task's name contains the string `fcm_make2*` and it does not have its own application configuration then [rose task-run](#) (page 267) will attempt to associate it with the corresponding `fcm_make*` application configuration.

### 11.1.3 Rose Configuration API

#### Rose App `fcm_make`

##### **Config args= ARG ...**

**Env Var** [ROSE\\_TASK\\_OPTIONS](#) (page 279)

For passing extra options and arguments to the `fcm make` command.

##### **Config dest-orig= PATH**

Specify the path to the destination of the original make. This is normally specified as a relative path under `$ROSE_SUITE_DIR/`. The default is `share/$ROSE_TASK_NAME`. If [rose task-run](#) (page 267) is invoked in `--new` mode, the application will remove this directory before running `fcm make --new`.

##### **Config dest-cont= PATH**

Specify the path to the destination of the continuation make. This is normally specified as a relative path under `$ROSE_SUITE_DIR/`. The default is to use the same location as [fcm\\_make/dest-orig](#) (page 284). If [rose task-run](#) (page 267) is invoked in `--new` mode, the application will remove this directory before running `fcm make --new`. (If this location is in the same physical location of the destination of the original make, you should only invoke [rose task-run](#) (page 267) `--new` on the original make. Otherwise, contents generated by the original make will be wiped clean before the continuation make begins.)

##### **Config fast-dest-root-orig= PATH**

Specify the path to an existing location that can be used as a fast working directory for the original make. If this is specified, the `fcm make` command will be invoked in a temporary directory under this location before being copied back to the actual destination.

##### **Config fast-dest-root-cont= PATH**

Specify the path to an existing location that can be used as a fast working directory for the continuation make. If this is specified, the `fcm make` command will be invoked in a temporary directory under this location before being copied back to the actual destination.

##### **Config make-name-orig= NAME**

Specify the context name of the original make. The default is a null string. You can specify an alternate context name if this is undesirable. The `fcm make` command will be invoked with the `--name=NAME` option of `fcm make`.

##### **Config make-name-cont= NAME**

Specify the context name of the continuation make. If the default `fcm_make → fcm_make2` mapping is used, the context name of the continuation make will be set to 2. You can specify an alternate context name if this is undesirable. The continuation command will be invoked with the `--name=NAME` option of `fcm make`.

##### **Config mirror-step= STEP-NAME**

Specify the name of the mirror step, if not mirror. The application will normally look for a matching task in the suite (e.g. `fcm_make → fcm_make2`) which will continue the `fcm make` command at a remote HOST. If such a task is found, it will add the configuration `mirror-target=HOST:cylc-run/$ROSE_SUITE_NAME/share/$ROSE_TASK_NAME` as an argument to the `fcm make` command to substitute the mirror target. To switch off this feature, set `STEP-NAME` to a null string, i.e. `mirror-step=`.

##### **Config opt.jobs= N**

**Env Var** [ROSE\\_TASK\\_N\\_JOBS](#) (page 279)

**Default** 4

This can be used to control the number of processes `fcm` make would use in parallel.

**Config `orig-cont-map= ORIG-NAME : CONT-NAME`**

This setting allows you to override the default `fcm_make` → `fcm_make2` mapping between the names of the original and the continuation tasks in the suite.

**Config `use-pwd= true`**

By default, the application changes the working directory to `$ROSE_SUITE_DIR/share/$ROSE_TASK_NAME`. This option will stop this, and the working directory is the normal working directory of the task.

## 11.2 rose\_ana

This built-in application runs the `rose-ana` analysis engine.

`rose-ana` performs various configurable analysis steps. For example a common usage is to compare two files and report whether they differ or not. It can write the details of any comparisons it runs to a database in the suite's log directory to assist with any automated updating of control data.

### 11.2.1 Invocation

In automatic selection mode, this built-in application will be invoked automatically if a task has a name that starts with `rose_ana*`.

### 11.2.2 Analysis Modules

The built-in application will search for suitable analysis modules to load firstly in the `ana` subdirectory of the `rose-ana` app, then in the `ana` subdirectory of the top-most suite directory. Any additional directories to search (for example a site-wide central directory) may be specified by setting the `rose.conf[rose-ana]method-path` (page 200) variable. Finally the `ana_builtin`s subdirectory of the Rose installation itself contains any built-in comparisons.

### 11.2.3 Configuration

The application configuration should contain configuration sections which describe an analysis step. These sections must follow a particular format:

- the name must begin with `ana::`. This is required for `rose-ana` to recognise it as a valid section.
- the next part gives the name of the class within one of the analysis modules, including namespace information; for example to use the built-in `FilePattern` class from the `grepper` module you would provide the name `grepper.FilePattern`.
- finally an expression within parentheses which may contain any string; this should be used to make comparisons using the same class unique, but can otherwise simply act as a description or note.

The content within each of these sections consists of a series of key-value option pairs, just like other standard Rose apps. However the availability of options for a given section is specified and controlled by the *class* rather than the meta-data. This makes it easy to provide your own analysis modules without requiring changes to Rose itself.

Therefore you should consult the documentation or source code of the analysis module you wish to use for details of which options it supports. Additionally, some special treatment is applied to all options depending on what they contain:

**Environment Variables** If the option contains any words prefixed by `$` they will be substituted for the equivalent environment variable, if one is available.

**Lists** If the option contains newlines it will be returned as a list of strings automatically.

**Argument substitution** If the option contains one or more pairs of braces ({} ) the option will be returned multiple times with the parentheses substituted once for each argument passed to [rose task-run](#) (page 267)

The app may also define a configuration section, [ana:config], whose key-value pairs define app-wide settings that are passed through to the analysis classes. In the same way that the task options are dependent on the class definition, interpretation of the config options is done by the class(es), so their documentation or source code should be consulted for details.

#### Rose App rose\_ana

##### Config [ana:config]

###### Config grepper-report-limit

Limits the number of lines printed when using the `rose.apps.ana_builtin.grepper` analysis class.

###### Config skip-if-all-files-missing

Causes the `rose.apps.ana_builtin.grepper` class to pass if all files to be compared are missing.

###### Config kgo-database

Turns on the [Rose Ana Comparison Database](#) (page 152).

##### Config [ana:ANALYSIS\_CLASS]

Define a new analysis step. ANALYSIS\_CLASS is the name of the analysis class e.g. `grepper`.  
`FilePattern`.

###### Config KEY= VALUE

Define an argument to ANALYSIS\_CLASS.

### 11.2.4 Analysis Classes

There is one built-in module of analysis classes called `grepper`.

**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.

**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.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.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).

## 11.3 rose\_arch

This built-in application provides a generic solution to configure site-specific archiving of suite files. It is designed to work under [rose task-run](#) (page 267).

The application is normally configured in a [rose-app.conf](#) (page 194). Global settings may be specified in an [rose\\_arch\[arch\]](#) (page 289) section. Each archiving target will have its own [arch:TARGET] section for specific settings, where TARGET would be a URI to the archiving location on your site-specific archiving system. Settings in a [arch:TARGET] section would override those in the global [rose\\_arch\[arch\]](#) (page 289) section for the given TARGET.

A target is considered compulsory, i.e. it must have at least one source, unless it is specified with the syntax [arch: (TARGET)]. In which case, TARGET is considered optional. The application will skip an optional target that has no actual source.

The application provides some useful functionalities:

- Incremental mode: store the archive target settings, checksums of source files and the return code of archive command. In a retry, it would only redo targets that did not succeed in the previous attempts.
- Rename source files.
- Tar-Gzip or Gzip source files before sending them to the archive.

### 11.3.1 Invocation

In automatic selection mode, this built-in application will be invoked automatically if a task has a name that starts with `rose_arch*`.

### 11.3.2 Example

```

# General settings
[arch]
command-format=foo put %(target)s %(sources)s
source-prefix=$ROSE_DATA/
target-prefix=foo://hello/

# Archive a file to a file
[arch:world.out]
source=hello/world.out

# Auto gzip (compression inferred from target extension)
[arch:planet.gz]
source=hello/planet.out

# Manual gzip (rose does not recognise the .out.gz extension)
[arch:planet.out.gz]
compress=gz
source=hello/planet.out

# Archive files matched by a glob to a directory
[arch:worlds/]
source=hello/worlds/*

# Archive multiple files matched by globs or names to a directory
[arch:worlds/]
source=hello/worlds/* greeting/worlds/* hi/worlds/*

# As above, but "greeting/worlds/*" may return an empty list
[arch:worlds/]
source=hello/worlds/* (greeting/worlds/*) hi/worlds/*

# Target is optional, implied that sources may all be missing
[arch:(black-box/)]
source=cats.txt dogs.txt

# Auto tar-gzip
[arch:galaxies.tar.gz]
source-prefix=hello/
source=galaxies/*
# File with multiple galaxies may be large, don't do its checksum
update-check=mtime+size

# Force gzip each source file
[arch:stars/]
source=stars/*
compress=gzip

# Source name transformation
[arch:moons.tar.gz]
source=moons/*
rename-format=%(cycle)s-%(name)s
source-edit-format=sed 's/Hello/Greet/g' %(in)s >%(out)s

# Source name transformation with a rename-parser
[arch:unknown/stuff.pax]
rename-format=hello/%(cycle)s-%(name_head)s%(name_tail)s
rename-parser=^ (?P<name_head>stuff) ing(?P<name_tail>-.*)$
source=stuffing-*.txt

```

### 11.3.3 Output

On completion, `rose_arch` (page 289) writes a status summary for each target to the standard output, which looks like this:

```
0 foo:///fred/my-su173/output0.tar.gz [compress=tar.gz]
+ foo:///fred/my-su173/output1.tar.gz [compress=tar.gz, t(init)=2012-12-
→02T20:02:20Z, dt(tran)=5s, dt(arch)=10s, ret-code=0]
+     output1/earth.txt (output1/human.txt)
+     output1/venus.txt (output1/woman.txt)
+     output1/mars.txt (output1/man.txt)
= foo:///fred/my-su173/output2.tar.gz [compress=tar.gz]
! foo:///fred/my-su173/output3.tar.gz [compress=tar.gz]
```

The first column is a status symbol, where:

- 0** An optional target has no real source, and is skipped.
- +** A target is added or updated.
- =** A target is not updated, as it was previously successfully updated with the same sources.
- !** Error updating this target.

If the first column and the second column are separated by a space character, the second column is a target. If the first column and the second column are separated by a tab character, the second column is a source in the target above.

For a target line, the third column contains the compress scheme, the initial time, the duration taken to transform the sources, the duration taken to run the archive command and the return code of the archive command. For a source line, the third column contains the original name of the source.

### 11.3.4 Configuration

#### Rose App `rose_arch`

**Config [arch] (alternate: arch:TARGET)**

**Config command-format= FORMAT**  
**Compulsory** True

A Pythonic `printf`-style format string to construct the archive command. It must contain the placeholders `% (sources)s` and `% (target)s` for substitution of the sources and the target respectively.

**Config compress= pax|tar|pax.gz|tar.gz|tgz|gz**

If specified, compress source files to the given scheme before sending them to the archive. If not specified, the compress scheme is automatically determined by the file extension of the target, if it matches one of the allowed values. For the `pax|tar` scheme, the sources will be placed in a TAR archive before being sent to the target. For the `pax.gz|tar.gz|tgz` scheme, the sources will be placed in a TAR-GZIP file before being sent to the target. For the `gz` scheme, each source file will be compressed by GZIP before being sent to the target.

**Config rename-format**

If specified, the source files will be renamed according to the specified format. The format string should be a Pythonic `printf`-style format string. It may contain the placeholder `% (cycle)s` (for the current `ROSE_TASK_CYCLE_TIME` (page 278), the placeholder `% (name)s` for the name of the file, and/or named placeholders that are generated by `rename-parser` (page 289).

**Config rename-parser**

Ignored if `rename-format` is not specified. Specify a regular expression to parse the name of a source. The regular expression should do named captures of strings from source file names,

which can then be used to substitute named placeholders in the corresponding [rename-format](#) (page 289).

**Config source= NAME**

**Compolsory** True

Specify a list of space delimited source file names and/or globs for matching source file names. (File names with space or quote characters can be escaped using quotes or backslashes, like in a shell.) Paths, if not absolute (beginning with a /), are assumed to be relative to [ROSE\\_SUITE\\_DIR](#) (page 278) or to \$ROSE\_SUITE\_DIR/PREFIX if [source-prefix](#) (page 290) is specified. If a name or glob is given in a pair of brackets, e.g. (hello-world.\*), the source is considered optional and will not cause a failure if it does not match any source file names. However, a compulsory target that ends up with no matching source file will be considered a failure.

**Config source-edit-format= FORMAT**

A Pythonic printf-style format string to construct a command to edit or modify the content of source files before archiving them. It must contain the placeholders % (in)s and % (out)s for substitution of the path to the source file and the path to the modified source file (which will be created in a temporary working directory).

**Config source-prefix= PREFIX**

Add a prefix to each value in a source declaration. A trailing slash should be added for a directory. Paths are assumed to be relative to [ROSE\\_SUITE\\_DIR](#) (page 278). This setting serves two purposes. It provides a way to avoid typing the same thing repeatedly. It also modifies the namespaces of the sources if the target is in a TAR or TAR-GZIP file. In the absence of this setting, the name of a source in a TAR or TAR-GZIP file is the path relative to [ROSE\\_SUITE\\_DIR](#) (page 278). By specifying this setting, the source names in a TAR or TAR-GZIP file will be shortened by the prefix.

**Config target-prefix= PREFIX**

Add a prefix to each target declaration. This setting provides a way to avoid typing the same thing repeatedly. A trailing slash (or whatever is relevant for the archiving system) should be added for a directory.

**Config update-check= mtime+size|md5|sha1|...**

Specify the method for checking whether a source has changed since the previous run. If the value is mtime+size, the application will use the modified time and size of the source, which is useful for large files, but is less correct. Otherwise, the value, if specified, should be the name of a hash object in Python's [hashlib](#) (<https://docs.python.org/2/library/hashlib.html>), such as md5 (default), sha1, etc. In this mode, the application will use the checksum (based on the specified hashing method) of the content of each source file to determine if it has changed or not.

## 11.4 rose\_bunch

For the running of multiple command variants in parallel under a single job, as defined by the application configuration.

Each variant of the command is run in the same working directory with its output directed to separate .out and .err files of the form:

```
bunch.<name>.out
```

Should you need separate working directories you should configure your command to create the appropriate subdirectory for working in.

---

**Note:** 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 e.g. if one command would require 1GB memory and you have configured your app to run up to four commands at once then you will need to request 4GB of memory.

---

### 11.4.1 Example

```
meta=rose_bunch
mode=rose_bunch

[bunch]
command-format#echo arg1: %(arg1)s, arg2: %(arg2)s, command-instance: %(command-
→instances)s
command-instances = 4
fail-handle = abort
incremental = True
names = fool bar2 baz3 qux4
pool-size=2

[bunch-args]
arg1=1 2 3 4
arg2=foo bar baz qux
```

### 11.4.2 Configuration

The application is normally configured in the [rose\\_bunch\[bunch\]](#) (page 291) and [rose\\_bunch\[bunch-args\]](#) (page 292) sections of the [rose-app.conf](#) (page 194) file.

#### Rose App rose\_bunch

##### **Config [bunch]**

###### **Config command-format= FORMAT** Compulsory True

A Pythonic `printf`-style format string to construct the commands to run. Insert placeholders `%(argname)s` for substitution of the arguments specified under [\[bunch-args\]](#) (page 292) to the invoked command. The placeholder `%(command-instances)s` is reserved for inserting an automatically generated index for the command invocation when using the `command-instances` setting.

###### **Config command-instances= N**

Allows the user to specify an integer value for the number of instances of a command they want to run. This generates the values used by the `%(command-instances)s` value in `command-format`. Useful for cases where the only difference between invocations would be an index number e.g. ensemble members. Note indexes start at 0.

###### **Config pool-size= N**

Allows the user to limit the number of concurrently running commands. If not specified then all command variations will be run at the same time.

###### **Config fail-mode= continue|abort** Default continue

Specify what action you want the job to take on the failure of a command that it is trying to run. If set to continue all command variants will be run by the job and the job will return a non-zero exit code upon completion e.g. if three commands are to be run and the second one fails, all three will be run and the job will exit with a return code of 1. Alternatively, if [fail-mode](#) (page 291) is set to abort then on failure of any one of the command variants it will stop trying to run any further variants N.B. the job will wait for any already running commands to finish before exiting. Commands that won't be run due to aborting will be reported in the job output with a `[SKIP]` prefix when running in verbose mode. For example in the case of three command variants with a `pool-size` (page 291) of 1 and [fail-mode=abort](#) (page 291), if the second variant failed then the job would exit with a non-zero error code without having run the third variant.

###### **Config incremental= true|false**

**Default** true

If set to `true` then only failed commands will be re-run on retrying running of the job. If any changes are made to the configuration being run then all variants will be re-run. Similarly, running the app with the `--new` option to [rose task-run](#) (page 267) will result in all commands being run. In verbose mode the app will report commands that won't be run due to previous successes in the job output with a `[PASS]` prefix.

**Config names= name1 name2 ...**

Allows defining names for each of the command variants to be run, facilitating identification in logs. If not set then commands will be identified by their index. The number of entries in the names must be the same as the number of entries in each of the args to be used.

**Config [bunch-args]**

This section is used to specify the various combinations of args to be passed to the [rose-app.conf](#) (`[command]` (page 194) specified under [\[bunch\] command-format](#) (page 291).

**Config argname= val1 val2 ...**

Allows defining named lists of argument values to pass to [\[bunch\] command-format](#) (page 291). Multiple named sets of arguments can be defined. Each argname can be referenced in the using `%({argname})`s. The only disallowed name is `command-instances`, which is reserved for the auto-generated list of instances when the [\[bunch\] command-instances=N](#) (page 291) option is used.

## 11.5 rose\_prune

A framework for housekeeping a cycling suite.

It prunes files and directories generated by suite tasks. It is designed to work under [rose task-run](#) (page 267) on the host that runs the suite daemon.

The application is normally configured in the [rose\\_prune\[prune\]](#) (page 293) section in the [rose-app.conf](#) (page 194) file.

All settings are expressed as a space delimited list of cycles, normally as *cycle points* or *offsets* relative to the current cycle. For [datetime cycling](#), the format of a cycle point should be an [ISO8601 datetime](#), and an offset should be an [ISO8601 duration](#). E.g. `-P1DT6H` is 1 day and 6 hours before the current cycle point.

The cycles of some settings also accept an optional argument followed by a colon. In these, the argument should be globs for matching items in the directory. If two or more globs are required, they should be separated by a space. In which case, either the argument should be quoted or the space should be escaped by a backslash.

### 11.5.1 Invocation

In automatic selection mode, this built-in application will be invoked automatically if a task has a name that starts with `rose_prune*`.

### 11.5.2 Example

```
meta=rose_prune
mode=rose_prune

[prune]
cycle-format{cycle_year_month}=CCYYMM
prune-remote-logs-at=-PT6H
archive-logs-at=-P1D
prune-server-logs-at=-P7D
prune{work}=-PT6H:task_x* -PT12H:*/other*.dat -PT18H:task_y* -PT24H
```

(continues on next page)

(continued from previous page)

```
prune{share}=-P1D:hello-*-(cycle)s.txt -P3M:monthly/%(cycle_year_month)s/
prune{share/cycle}=-PT6H:foo* -PT12H:'bar* *.baz*' -P1D
```

### 11.5.3 Configuration

**Rose App rose\_prune**

#### **Config [prune]**

##### **Config cycle-format{key}= format**

Specify a key to a format string for use in conjunction with a :rose:conf‘prune{item-root}=cycle:globs’ setting. For example, we may have something like cycle-format{cycle\_year}=CCYY and prune{share}=-P1Y:xmas-present-%(cycle\_year)s/. In Cyc, if the current cycle point is 20151201T0000Z, it will clear out the directory share/xmas-present-2014/.

The key can be any string that can be used in a % (key) s substitution, and format should be a valid [rose date](#) (page 250) print format.

##### **Config prune-remote-logs-at= cycle ...**

Re-sync remote job logs at these cycles and remove them from remote hosts.

##### **Config prune-server-logs-at= cycle ...**

Remove logs on the suite server. Removes both log directories and archived logs.

##### **Config archive-logs-at= cycle ...**

Archive all job logs at these cycles. Remove remote job logs on success.

##### **Config prune{item-root}= cycle[:globs] ...**

Remove the sub-directories under item-root (e.g. [work](#)) of the specified cycles. E.g. In Cyc, if current cycle is 20141225T1200Z, prune{work}=-PT12H will clear out work/20141225T0000Z/.

If globs are specified for a cycle, it will attempt to prune only items matching CYCLE/GLOBS under item-root. E.g. In Cyc, if current cycle is 20141225T1200Z, then prune{share/cycle}=-PT12H:wild\* will clear out all items matching share/cycle/20141225T0000Z/wild\*.

A glob can also be specified as a formatting string containing a single substitution %(cycle)s. In this mode, the cycle string will not be added as a sub-directory of the item-root. E.g. In Cyc, if current cycle is 20141225T1200Z, then prune{share}=-PT12H:hello-\*-(cycle)s.txt will clear out all items matching share/hello-\*-(20141225T1200Z).txt.

A glob can also be specified as a formatting string containing a substitution % (key) s, if a [cycle-format {key}=format](#) (page 293) setting is specified.

##### **Config prune-work-at= cycle[:globs] ...**

Deprecated since version foo: Equivalent to prune{work}=cycle[:globs] ....

##### **Config prune-datac-at= cycle[:globs] ...**

Deprecated since version foo: Equivalent to prune{share/cycle}=cycle[:globs] ..

..

foo



## ROSE GTK LIBRARY

The Rose/Rosie GUIs (such as the config editor) are written using the Python bindings for the GTK GUI toolkit ([PyGTK](http://www.pygtk.org/) (<http://www.pygtk.org/>)). You can write your own custom GTK widgets and use them within Rose. They should live with the metadata under the `lib/python/widget/` directory.

### 12.1 Value Widgets

Value widgets are used for operating on the values of settings. In the config editor, they appear next to the menu button and name label. There are builtin value widgets in Rose such as text entry boxes, radio buttons, and drop-down menus. These are chosen by the config editor based on metadata - for example, if a setting has an integer type, the value widget will be a spin button.

The config editor supports adding user-defined custom widgets which replace the default widgets. These have the same API, but live in the metadata directories rather than the Rose source code.

For example, you may wish to add widgets that deal with dates (e.g. using something based on a [calendar](https://developer.gnome.org/pygtk/stable/class-gtkcalendar.html) (<https://developer.gnome.org/pygtk/stable/class-gtkcalendar.html>) widget) or use a [slider](http://www.pygtk.org/pygtk2tutorial/sec-RangeWidgetExample.html) (<http://www.pygtk.org/pygtk2tutorial/sec-RangeWidgetExample.html>) widget for numbers. You may even want something that uses an image-based interface such as a latitude-longitude chooser based on a map.

Normally, widgets will be placed within the metadata directory for the suite or application. Widgets going into the Rose core should be added to the `lib/python/rose/config_editor/valuewidget/` directory in a Rose distribution.

#### 12.1.1 Example

See the [Advanced Tutorial](#) (page 170).

#### 12.1.2 API Reference

All value widgets, custom or core, use the same API. This means that a good practical reference is the set of existing value widgets in the package `rose.config_editor.valuewidget`.

The procedure for implementing a custom value widget is as follows:

Assign a `widget [rose-config-edit]` attribute to the relevant variable in the metadata configuration, e.g.

```
[namelist:VerifConNL/ScalarAreaCodes]
widget [rose-config-edit]=module_name.AreaCodeChooser
```

where the widget class lives in the module `module_name` under `lib/python/widget/` in the metadata directory for the application or suite. Modules are imported by the config editor on demand.

This class should have a constructor of the form

```
class AreaCodeChooser(Gtk.HBox):
    def __init__(self, value, metadata, set_value, hook, arg_str=None)
```

with the following arguments:

**value** a string that represents the value that the widget should display.

**metadata** a map or dictionary of configuration metadata properties for this value, such as

```
{'type': 'integer', 'help': 'This is used to count something'}
```

---

**Note:** You may not need to use this information.

---

**set\_value** a function that should be called with a new string value of this widget, e.g.

```
set_value("20")
```

**hook** An instance of a class `rose.config_editor.valuewidget.ValueWidgetHook` containing callback functions that you should connect some of your widgets to.

**arg\_str** a keyword argument that stores extra text given to the widget option in the metadata, if any:

```
widget[rose-config-edit]=modulename.ClassName arg1 arg2 arg3 ...
```

would give a `arg_str` of "arg1 arg2 arg3 ...". This could help configure your widget - for example, for a table based widget, you might give the column names:

```
widget[rose-config-edit]=table.TableValueWidget NAME ID WEIGHTING
```

This means that you can write a generic widget and then configure it for different cases.

`hook` contains some callback functions that you should implement:

**hook.get\_focus(widget) -> None** which you should connect your top-level widget (`self`) to as follows:

```
self.grab_focus = lambda: hook.get_focus(my_favourite_focus_widget)
```

or define a method in your class

```
def grab_focus(self):
    """Override the focus method, so we can scroll to a particular widget."""
    return hook.get_focus(my_favourite_focus_widget)
```

which allows the correct widget (`my_favourite_focus_widget`) in your container to receive the focus such as a `Gtk.Entry` (`my_favourite_focus_widget`) and will also trigger a scroll action on a config editor page. This is important to implement to get the proper global find functionality.

**hook.trigger\_scroll(widget) -> None** accessed by

```
hook.trigger_scroll(my_favourite_focus_widget)
```

This should be connected to the `focus-in-event` GTK signal of your top-level widget (`self`):

```
self.entry.connect('focus-in-event',
                  hook.trigger_scroll)
```

This also is used to trigger a config editor page scroll to your widget.

You may implement the following optional methods for your widget, which help to preserve cursor position when a widget is refreshed:

---

**set\_focus\_index(focus\_index) -> None** A method that takes a number as an argument, which is the current cursor position relative to the characters in the variable value:

```
def set_focus_index(self, focus_index):
    """Set the cursor position to focus_index."""
    self.entry.set_position(focus_index)
```

For example, a `focus_index` of 0 means that your widget should set the cursor position to the beginning of the value. A `focus_index` of 4 for a variable value of `Operational` means that the cursor should be placed between the `r` and the `a`.

---

**Note:** This has no real meaning or importance for widgets that don't display editable text. If you do not supply this method, the config editor will attempt to do the right thing anyway.

**get\_focus\_index() -> focus\_index** A method that takes no arguments and returns a number which is the current cursor position relative to the characters in the variable value:

```
def get_focus_index(self):
    """Return the cursor position."""
    return self.entry.get_position()
```

---

**Note:** This has no real meaning or importance for widgets that don't display editable text. If you do not supply this method, the config editor will guess the cursor position anyway, based on the last change to the variable value.

**handle\_type\_error(is\_in\_error) -> None** The default behaviour when a variable error is added or removed is to re-instantiate the widget (refresh and redraw it). This can be overridden by defining this method in your value widget class. It takes a boolean `is_in_error` which is `True` if there is a value (type) error and `False` otherwise:

```
def handle_type_error(self, is_in_error):
    """Change behaviour based on whether the variable is_in_error."""
    icon_id = gtk STOCK_DIALOG_ERROR if is_in_error else None
    self.entry.set_icon_from_stock(0, gtk STOCK_DIALOG_ERROR)
```

For example, this is used in a built-in widget for the quoted string types `string` and `character`. The quotes around the text are normally hidden, but the `handle_type_error` shows them if there is an error. The method also keeps the keyboard focus, which is the main purpose.

---

**Tip:** You may not have much need for this method, as the default error flagging and cursor focus handling is normally sufficient.

---

**Tip:** All the existing variable value widgets are implemented using this API, so a good resource is the modules within the `lib/python/rose/config_editor/valuewidget` package.

## 12.2 Config Editor Custom Pages

A ‘page’ in the config editor is the container inside a tab or detached tab that (by default) contains a table of variable widgets. The config editor allows custom ‘pages’ to be defined that may or may not use the standard set of variable widgets (menu button, name, value widget). This allows any presentation of the underlying variable information.

For example, you may wish to present the variables in a more structured, two-dimensional form rather than as a simple list. You may want to strip down or add to the information presented by default - e.g. hiding names or embedding widgets within a block of help text.

You may even wish to do something off-the-wall such as an `xdot`-based (<https://github.com/jrfonseca/xdot.py>) widget set!

### 12.2.1 API Reference

The procedure for generating a custom page widget is as follows:

Assign a `widget` option to the relevant namespace in the metadata configuration, e.g.

```
[ns:namelist/STASHNUM]
widget [rose-config-edit]=module_name.MyGreatBigTable
```

The widget class should have a constructor of the form

```
class MyGreatBigTable(gtk.Table):

    def __init__(self, real_variable_list, missing_variable_list,
                 variable_functions_inst, show_modes_dict,
                 arg_str=None):
```

The class can inherit from any `gtk.Container`-derived class.

The constructor arguments are

**real\_variable\_list** a list of the `Variable` objects (`x.name`, `x.value`, `x.metadata`, etc from the `rose.variable` module). These are the objects you will need to generate your widgets around.

**missing\_variable\_list** a list of ‘missing’ `Variable` objects that could be added to the container. You will only need to worry about these if you plan to show them by implementing the ‘View Latent’ menu functionality that we’ll discuss further on.

**variable\_functions\_inst** an instance of the class `rose.config_editor.ops.variable.VariableOperations`. This contains methods to operate on the variables. These will update the undo stack and take care of any errors. These methods are the only ways that you should write to the variable states or values. For documentation, see the module `lib/python/rose/config_editor/ops/variable.py`.

**show\_modes\_dict** a dictionary that looks like this:

```
show_modes_dict = {'latent': False, 'fixed': False, 'ignored': True,
                   'user-ignored': False, 'title': False,
                   'flag:optional': False, 'flag:no-meta': False}
```

which could be ignored for most custom pages, as you need. The meaning of the different keys in a non-custom page is:

**'latent'** False means don’t display widgets for variables in the metadata or that have been deleted (the `variable_list.ghosts` variables)

**'fixed'** False means don’t display widgets for variables if they only have one value set in the metadata `values` option.

**'ignored'** False means don’t display widgets for variables if they’re ignored (in the configuration, but commented out).

**'user-ignored'** (If `ignored` is False) False means don’t display widgets for user-ignored variables. True means always show user-ignored variables.

**'title'** Short for ‘View with no title’, False means show the title of a variable, True means show the variable name instead.

**'flag:optional'** True means indicate if a variable is optional, and False means do not show an indicator.

**'flag:no-meta'** True means indicate if a variable has any metadata content, and False means do not show an indicator.

If you wish to implement actions based on changes in these properties (e.g. displaying and hiding fixed variables depending on the 'fixed' setting), the custom page widget should expose a method named 'show\_mode\_change' followed by the key. However, 'ignored' is handled separately (more below). These methods should take a single boolean that indicates the display status. For example:

```
def show_fixed(self, should_show)
```

The argument `should_show` is a boolean. If True, fixed variables should be shown. If False, they should be hidden by your custom container.

**arg\_str** a keyword argument that stores extra text given to the `widget` option in the metadata, if any:

```
widget[rose-config-edit] = modulename.ClassName arg1 arg2 arg3 ...
```

would give a `arg_str` of "arg1 arg2 arg3 ...". This could help configure your widget - for example, for a table based widget, you might give the column names:

```
widget[rose-config-edit] = table.TableValueWidget NAME ID WEIGHTING
```

This means that you can write a generic widget and then configure it for different cases.

Refreshing the whole page in order to display a small change to a variable (the default) can be undesirable. To deal with this, custom page widgets can optionally expose some variable-change specific methods that do this themselves. These take a single `rose.variable.Variable` instance as an argument.

**add\_variable\_widget** (`self, variable`) → None

Will be called when a variable is created.

**reload\_variable\_widget** (`self, variable`) → None

Will be called when a variable's status is changed, e.g. it goes into an error state.

**remove\_variable\_widget** (`self, variable`) → None

Will be called when a variable is removed.

**update\_ignored** (`self`) → None

Will be called to allow you to update ignored widget display, if (for example) you show/hide ignored variables. If you don't have any custom behaviour for ignored variables, it's worth writing a method that does nothing - e.g. one that contains just `pass`).

If you take the step of using your own variable widgets, rather than the `VariableWidget` class in `lib/python/rose/config_editor/variable.py` (the default for normal config-edit pages), each variable-specific widget should have an attribute `variable` set to their `rose.variable.Variable` instance. You can implement 'ignored' status display by giving the widget a method `set_ignored` which takes no arguments. This should examine the `ignored_reason` dictionary attribute of the widget's `variable` instance - the variable is ignored if this is not empty. If the variable is ignored, the widget should indicate this e.g. by greying out part of it.

---

**Tip:** All existing page widgets use this API, so a good resource is the modules in `lib/python/rose/config_editor/pagewidget/`.

---

Generally speaking, a visible change, click, or key press in the custom page widget should make instant changes to variable value(s), and the value that the user sees. Pages are treated as temporary, superficial views of variable data, and changes are always assumed to be made directly to the main copy of the configuration in memory (this is automatic when the `rose.config_editor.ops.variable.VariableOperations` methods are used, as they should be). Closing the page shouldn't change, or lose, any data! The custom class should return a `gtk` object to be packed into the page framework, so it's best to subclass from an existing `gtk` Container type such as `gtk.VBox` (or `gtk.Table`, in the example above).

---

**Note:** In line with the general philosophy, metadata should not be critical to page operations - it should be capable of displaying variables even when they have no or very little metadata, and still make sense if some variables are missing or new.

---

## 12.3 Config Editor Custom Sub Panels

A ‘sub panel’ or ‘summary panel’ in the config editor is a panel that appears at the bottom of a page and is intended to display some summarised information about sub-pages (sub-namespaces) underneath the page. For example, the top-level file page, by default, has a sub panel to summarise the individual file sections.

Any actual data belonging to the page will appear above the sub panel in a separate representation.

Sub panels are capable of using quite a lot of functionality such as modifying the sections and options in the sub-pages directly.

### 12.3.1 API Reference

The procedure for generating a custom sub panel widget is as follows:

Assign a `widget [rose-config-edit:sub-ns]` option to the relevant namespace in the metadata configuration, e.g.

```
[ns:namelist/all_the_foo_namelists]
widget [rose-config-edit:sub-ns]=module_name.MySubPanelForFoos
```

Note that because the actual data on the page has a separate representation, you need to write `[rose-config-edit:sub-ns]` rather than just `[rose-config-edit]`.

The widget class should have a constructor of the form

```
class MySubPanelForFoos(Gtk.VBox):
    def __init__(self, section_dict, variable_dict,
                 section_functions_inst, variable_functions_inst,
                 search_for_id_function, sub_functions_inst,
                 is_duplicate_boolean, arg_str=None):
```

The class can inherit from any `Gtk.Container`-derived class.

The constructor arguments are:

**section\_dict** a dictionary (map, hash) of section name keys and section data object values (instances of the `rose.section.Section` class). These contain some of the data such as section ignored status and comments that you may want to present. These objects can usually be used by the `section_functions_inst` methods as arguments - for example, passed in in order to ignore or enable a section.

**variable\_dict** a dictionary (map, hash) of section name keys and lists of variable data objects (instances of the `rose.variable.Variable` class). These contain useful information for the variable (option) such as state, value, and comments. Like section data objects, these can usually be used as arguments to the `variable_functions_inst` methods to accomplish things like changing a variable value or adding or removing a variable.

**section\_functions\_inst** an instance of the class `rose.config_editor.ops.section.SectionOperations`. This contains methods to operate on the variables. These will update the undo stack and take care of any errors. Together with `sub_functions_inst`, these methods are the only ways that you should write to the section states or other attributes. For documentation, see the module `lib/python/rose/config_editor/ops/section.py`.

**variable\_functions\_inst** an instance of the class `rose.config_editor.ops.variable.VariableOperations`. This contains methods to operate on the variables. These will update the undo stack and take care of any errors. These methods are the only ways that you should write to the variable states or values. For documentation, see the module `lib/python/rose/config_editor/ops/variable.py`.

**search\_for\_id\_function** a function that accepts a setting id (a section name, or a variable id) as an argument and asks the config editor to navigate to the page for that setting. You could use this to allow a click on a section name in your widget to launch the page for the section.

**sub\_functions\_inst** an instance of the class `rose.config_editor.ops.group.SubDataOperations`. This contains some convenience methods specifically for sub panels, such as operating on many sections at once in an optimised way. For documentation, see the module `lib/python/rose/config_editor/ops/group.py`.

**is\_duplicate\_boolean** a boolean that denotes whether or not the sub-namespaces in the summary data consist only of duplicate sections (e.g. only `namelist:foo(1)`, `namelist:foo(2)`, ...). For example, this could be used by your widget to decide whether to implement a “Copy section” user option.

**arg\_str** a keyword argument that stores extra text given to the `widget` option in the metadata, if any - e.g.:

```
widget[rose-config-edit:sub-ns] = modulename.ClassName arg1 arg2 arg3 ...
```

would give a `arg_str` of "arg1 arg2 arg3 ...". You can use this to help configure your widget.

---

**Tip:** All existing sub panel widgets use this API, so a good resource is the modules in `lib/python/rose/config_editor/panelwidget/`.

---



## ROSE MACRO API

Rose macros manipulate or check configurations, often based on their metadata. There are four types of macros:

**Checkers (validators)** Check a configuration, perhaps using metadata.

**Changers (transformers)** Change a configuration e.g. adding/removing options.

**Upgraders** Special transformer macros for upgrading and downgrading configurations (covered in the *Upgrade Macro API* (page 309)).

**Reporters** output information about a configuration.

They can be run within *rose config-edit* (page 250) or via *rose macro* (page 254).

---

**Note:** This section covers validator, transformer and reporter macros. For upgrader macros see *Upgrade Macro API* (page 309).

---

There are built-in Rose macros that handle standard behaviour such as trigger changing and type checking.

Macros use a Python API, and should be written in Python, unless you are doing something very fancy. In the absence of a Python house style, it's usual to follow the standard Python style guidance ([PEP8](#) (<https://www.python.org/dev/peps/pep-0008/>), [PEP257](#) (<https://www.python.org/dev/peps/pep-0257/>))).

---

**Tip:** You should avoid writing validator macros if the checking can be expressed via *metadata* (page 208).

---

### 13.1 Location

A module containing macros should be stored under a directory `lib/python/macros/` in the metadata for a configuration. This directory should be a Python package.

When developing macros for Rose internals, macros should be placed in the `rose.macros` package in the Rose Python library. They should be referenced by the `lib/python/rose/macros/__init__.py` classes and a call to them can be added in the `lib/python/rose/config_editor/main.py` module if they need to be run implicitly by the config editor.

### 13.2 Writing Macros

---

**Note:** For basic usage see the *macro tutorial* (page 126).

---

Validator, transformer and reporter macros are Python classes which subclass from `rose.macro.MacroBase` (page 305) ([API](#) (page 305)).

These macros implement their behaviours by providing a validate, transform or report method. A macro can contain any combination of these methods so, for example, a macro might be both a validator and a transformer.

These methods should accept two `rose.config.ConfigNode` (page 232) instances as arguments - one is the configuration, and one is the metadata configuration that provides information about the configuration items.

---

**Tip:** See also [Python](#) (page 228).

---

A validator macro should look like:

```
import rose.macro

class SomeValidator(rose.macro.MacroBase):

    """This does some kind of check."""

    def validate(self, config, meta_config=None):
        # Some check on config appends to self.reports using self.add_report
        return self.reports
```

The returned list should be a list of `rose.macro.MacroReport` (page 307) objects containing the section, option, value, and warning strings (info) for each setting that is in error. These are initialised behind the scenes by calling the inherited method `rose.macro.MacroBase.add_report()` (page 306) via `self.add_report()`. This has the form:

```
def add_report(self, section=None, option=None, value=None, info=None,
              is_warning=False):
```

This means that you should call it with the relevant section first, then the relevant option, then the relevant value, then the relevant error message, and optionally a warning flag that we'll discuss later. If the setting is a section, the option should be `None` and the value `None`. For example,

```
def validate(self, config, meta_config=None):
    editor_value = config.get(["env", "MY_FAVOURITE_STREAM_EDITOR"]).value
    if editor_value != "sed":
        self.add_report("env",                         # Section
                        "MY_FAVOURITE_STREAM_EDITOR", # Option
                        editor_value,                # Value
                        "Should be 'sed'!")         # Message
    return self.reports
```

Validator macros have the option to give warnings, which do not count as formal errors in the Rose config editor GUI. These should be used when something *may* be wrong, such as warning when using an advanced-developer-only option. They are invoked by passing a 5th argument to `self.add_report()`, `is_warning`, like so:

```
self.add_report("env",
                "MY_FAVOURITE_STREAM_EDITOR",
                editor_value,
                "Could be 'sed!'",
                is_warning=True)
```

A transformer macro should look like:

```
import rose.macro

class SomeTransformer(rose.macro.MacroBase):

    """This does some kind of change to the config."""
```

(continues on next page)

(continued from previous page)

```
def transform(self, config, meta_config=None):
    # Some operation on config which calls self.add_report for each change.
    return config, self.reports
```

The returned list should be a list of 4-tuples containing the section, option, value, and information strings for each setting that was changed (e.g. added, removed, value changed). If the setting is a section, the option should be `None` and the value `None`. If an option was removed, the value should be the old value - otherwise it should be the new one (added/changed). For example,

```
def transform(self, config, meta_config=None):
    """Add some more snow control."""
    if config.get(["namelist:snowflakes"]) is None:
        config.set(["namelist:snowflakes"])
        self.add_report(list_of_changes,
                        "namelist:snowflakes", None, None,
                        "Updated snow handling in time for Christmas")
    config.set(["namelist:snowflakes", "l_unique"], ".true.")
    self.add_report("namelist:snowflakes", "l_unique", ".true.",
                   "So far, anyway.")
    return config, self.reports
```

The current working directory within a macro is always the configuration's directory. This makes it easy to access `non-rose-app.conf` files (e.g. in the `file/` subdirectory).

There are also reporter macros which can be used where you need to output some information about a configuration. A reporter macro takes the same form as validator and transform macros but does not require a return value.

```
def report(self, config, meta_config=None):
    """ Write some information about the configuration to a report file.

    Note: report methods do not have a return value.

    """
    with open('report/file', 'r') as report_file:
        report_file.write(str(config.get(["namelist:snowflakes"])))
```

Macros also support the use of keyword arguments, giving you the ability to have the user specify some input or override to your macro. For example a transformer macro could be written as follows to allow the user to input `some_value`:

```
def transform(self, config, meta_config=None, some_value=None):
    """Some transformer macro"""
    return
```

---

**Note:** The extra arguments require default values (`=None` in this example) and that you should add error handling for the input accordingly.

---

On running your macro the user will be prompted to supply values for these arguments or accept the default values.

### 13.3 Python API

Module to list or run available custom macros for a configuration.

It also stores macro base classes and macro library functions.

```
class rose.macro.MacroBase
    Base class for macros for validating or transforming configurations.
```

**Synopsis:**

```
>>> import rose.macro
...
>>> class SomeValidator(rose.macro.MacroBase):
...
...     '''Important: Add a docstring for your macro like this.
...
...     A macro class should implement one of the following methods:
...
...     ...
...
...     def validate(self, config, meta_config=None):
...         # Some check on config appends to self.reports using
...         # self.add_report.
...         return self.reports
...
...     def transform(self, config, meta_config=None):
...         # Some operation on config which calls self.add_report
...         # for each change.
...         return config, self.reports
...
...     def report(self, config, meta_config=None):
...         # Perform some analysis of the config but return nothing.
...         pass
```

**add\_report(\*args, \*\*kwargs)**

Add a rose.macro.MacroReport.

See [rose.macro.MacroReport](#) (page 307) for details of arguments.

**Examples**

```
>>> # An example validator macro which adds a report to the setting
>>> # env=MY_FAVOURITE_STREAM_EDITOR.
>>> class My_Macro(MacroBase):
...     def validate(self, config, meta_config=None):
...         editor_value = config.get(
...             ['env', 'MY_FAVOURITE_STREAM_EDITOR']).value
...         if editor_value != 'sed':
...             self.add_report(
...                 'env',                      # Section
...                 'MY_FAVOURITE_STREAM_EDITOR', # Option
...                 editor_value,               # Value
...                 'Should be "sed"!')        # Message
...         return self.reports
```

**get\_metadata\_for\_config\_id(setting\_id, meta\_config)**

Return a dict of metadata properties and values for a setting id.

**Parameters**

- **setting\_id** (*str*) – The name of the setting to extract metadata for.
- **meta\_config** ([rose.config.ConfigNode](#) (page 232)) – Config node containing the metadata to extract from.

**Returns** A dictionary containing metadata options.

**Return type** *dict*

## Example

```
>>> # Create a rose app.
>>> with open('rose-app.conf', 'w+') as app_config:
...     app_config.write('''
... [foo]
... bar=2
...
... ''')
>>> os.mkdir('meta')
>>> with open('meta/rose-meta.conf', 'w+') as meta_config:
...     meta_config.write('''
... [foo=bar]
... values = 1,2,3
...
... ''')
...
...
>>> # Load config.
>>> app_conf, config_map, meta_config = load_conf_from_file(
...     '.', 'rose-app.conf')
...
...
>>> # Extract metadata for foo=bar.
>>> get_metadata_for_config_id('foo=bar', meta_config)
{'values': '1,2,3', 'id': 'foo=bar'}
```

### `get_resource_path(filename=“”)`

Load the resource according to the path of the calling macro.

The returned path will be based on the macro location under `lib/python` in the metadata directory.

If the calling macro is `lib/python/macro/levels.py`, and the filename is `rules.json`, the returned path will be `etc/macro/levels/rules.json`.

**Parameters** `filename (str)` – The filename of the resource to request the path to.

**Returns** The path to the requested resource.

**Return type** str

### `pretty_format_config(config)`

Standardise the keys and values of a config node.

**Parameters** `config (rose.config.ConfigNode (page 232))` – The config node to convert.

### `standard_format_config(config)`

Standardise any degenerate representations e.g. namelist repeats.

**Parameters** `config (rose.config.ConfigNode (page 232))` – The config node to convert.

## `class rose.macro.MacroReport(section=None, option=None, value=None, info=None, is_warning=False)`

Class to hold information about a macro issue.

### Parameters

- `section (str)` – The name of the section to attach this report to.
- `option (str)` – The name of the option (within the section) to attach this report to.
- `value (obj)` – The value of the configuration associated with this report.
- `info (str)` – Text information describing the nature of the report.
- `is_warning (bool)` – If True then this report will be logged as a warning.

### Example

```
>>> report = MacroReport('env', 'WORLD', 'Earth',
...                         'World changed to Earth', True)
```

## ROSE UPGRADE MACRO API

Rose upgrade macros are used to upgrade application configurations between metadata versions. They are classes, very similar to [transformer macros](#) (page 303), but with a few differences:

- An `upgrade` method instead of a `transform` method
- An optional  `downgrade` method, identical in API to the `upgrade` method, but intended for performing the reverse operation
- A more helpful API via `rose.upgrade.MacroUpgrade` (page 309) methods
- `BEFORE_TAG` and `AFTER_TAG` attributes - the version of metadata they apply to (`BEFORE_TAG`) and the version they upgrade to (`AFTER_TAG`)

An example upgrade macro might 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):  
        self.add_setting(config, ["env", "NEW_VARIABLE"], "0")  
        self.remove_setting(config, ["namelist:old_things", "OLD_VARIABLE"])  
        return config, self.reports
```

---

**Note:** The class name is unimportant - the `BEFORE_TAG` and `AFTER_TAG` identify the macro.

---

Metadata versions are usually structured in a `rose-meta/CATEGORY/VERSION/` hierarchy - where `CATEGORY` denotes the type or family of application (sometimes it is the command used), and `VERSION` is the particular version e.g. `27.2` or `HEAD`.

Upgrade macros live under the `CATEGORY` directory in a `versions.py` file - `rose-meta/CATEGORY/versions.py`.

---

**Tip:** If you have many upgrade macros, you may want to separate them into different modules in the same directory. You can then import from those in `versions.py`, so that they are still exposed in that module. You'll need to make your directory a package by creating an `__init__.py` file, which should contain the line `import versions`. To avoid conflict with other `CATEGORY` upgrade modules (or other Python modules), please name these very modules carefully or use absolute or package level imports like this: `from .versionXX_YY import FooBar`.

---

Upgrade macros are subclasses of `rose.upgrade.MacroUpgrade` (page 309). They have all the functionality of the [transformer macros](#) (page 303).

**class** rose.upgrade.MacroUpgrade

Class derived from MacroBase to aid upgrade functionality.

**act\_from\_files**(config, downgrade=False)

Parse a change configuration into actions.

Searches for:

- etc/VERSION/rose-macro-add.conf (settings to be added)
- etc/VERSION/rose-macro-remove.conf (settings to be removed)

Where VERSION is equal to self.BEFORE\_TAG.

If settings are defined in either file, and changes can be made, the self.reports will be updated automatically.

Note that act\_from\_files can be used in combination with other methods as part of the same upgrade.

**Parameters**

- **config** (rose.config.ConfigNode (page 232)) – The application configuration.
- **downgrade** (bool) – True if downgrading.

**Returns** None**add\_setting**(config, keys, value=None, forced=False, state=None, comments=None, info=None)

Add a setting to the configuration.

**Parameters**

- **config** (rose.config.ConfigNode (page 232)) – The application configuration.
- **keys** (list) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.
- **value** (string – optional) – String denoting the new setting value. Required for options but not for settings.
- **forced** (bool – optional) – If True override value if the setting already exists.
- **state** (str – optional) – The state of the new setting - should be one of the rose.config.ConfigNode states e.g. rose.config.ConfigNode.STATE\_USER\_IGNORED. Defaults to rose.config.ConfigNode.STATE\_NORMAL.
- **comments** (list – optional) – List of comment lines (strings) for the new setting or None.
- **info** (string – optional) – A short string containing no new lines, describing the addition of the setting.

**Returns** None**change\_setting\_value**(config, keys, value, forced=False, comments=None, info=None)

Change a setting (option) value in the configuration.

**Parameters**

- **config** (rose.config.ConfigNode (page 232)) – The application configuration.
- **keys** (list) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.
- **value** (string) – The new value. Required for options, can be None for sections.

- **forced** (*bool*) – Create the setting if it is not present in config.
- **comments** (*list* – *optional*) – List of comment lines (strings) for the new setting or None.
- **info** (*string* – *optional*) – A short string containing no new lines, describing the addition of the setting.

**Returns** None

**enable\_setting** (*config, keys, info=None*)

Enable a setting in the configuration.

#### Parameters

- **config** ([rose.config.ConfigNode](#) (page 232)) – The application configuration.
- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.
- **info** (*string* – *optional*) – A short string containing no new lines, describing the addition of the setting.

**Returns** False - if the setting’s state is not changed else None.

**get\_setting\_value** (*config, keys, no\_ignore=False*)

Return the value of a setting or None if not set.

#### Parameters

- **config** ([rose.config.ConfigNode](#) (page 232)) – The application configuration.
- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.
- **no\_ignore** (*bool* – *optional*) – If True return None if the setting is ignored (else return the value).

**Returns** object - The setting value or None if not defined.

**ignore\_setting** (*config, keys, info=None, state='!'*)

User-ignore a setting in the configuration.

#### Parameters

- **config** ([rose.config.ConfigNode](#) (page 232)) – The application configuration.
- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.
- **info** (*string* – *optional*) – A short string containing no new lines, describing the addition of the setting.
- **state** (*string* – *optional*) – A [rose.config.ConfigNode](#) state. STATE\_USER\_IGNORED by default.

**Returns** False - if the setting’s state is not changed else None.

**remove\_setting** (*config, keys, info=None*)

Remove a setting from the configuration.

#### Parameters

- **config** ([rose.config.ConfigNode](#) (page 232)) – The application configuration.
- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.

- **info** (*string – optional*) – A short string containing no new lines, describing the addition of the setting.

**Returns** None

**rename\_setting** (*config, keys, new\_keys, info=None*)

Rename a setting in the configuration.

**Parameters**

- **config** ([rose.config.ConfigNode](#) (page 232)) – The application configuration.
- **keys** (*list*) – A list defining a hierarchy of node.value ‘keys’. A section will be a list of one keys, an option will have two.
- **new\_keys** (*list*) – The new hierarchy of node.value ‘keys’.
- **info** (*string – optional*) – A short string containing no new lines, describing the addition of the setting.

**Returns** None

## ROSIE WEB API

This section explains how to use the Rosie web service API. All Rosie discovery services (e.g. rosie search, rosie go, web page) use a RESTful ([https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)) API to interrogate a web server, which then interrogates an RDBMS ([https://en.wikipedia.org/wiki/Relational\\_database\\_management\\_system](https://en.wikipedia.org/wiki/Relational_database_management_system)). Returned data is encoded in the JSON (<http://www.json.org/>) format.

### 15.1 Location

The URLs to access the web API of a Rosie web service (with a given prefix name) can be found in your Rose site configuration file as the value of [rosie-id]prefix-ws.PREFIX\_NAME. To access the API for a given repository with prefix PREFIX\_NAME, you must select a format (the only currently supported format is JSON) and use a url that looks like:

```
http://host/<PREFIX_NAME>/get_known_keys?format=json
```

### 15.2 REST API

#### GET (str: prefix) /get\_known\_keys

Return the main property names stored for suites (e.g. idx, branch, owner) plus any additional names specified in the site config.

##### Parameters

- **prefix** (str) – Repository prefix.
- **format** (string) – Desired return format (json or None).

##### Example Request

```
GET http://host/my_prefix/get_known_keys?format=json HTTP/1.1
```

##### Example Response

```
["access-list", "idx", "branch", "owner", "project", "revision", "status",  
 ↴ "title"]
```

#### GET (str: prefix) /get\_optional\_keys

Return all unique optional or user-defined property names given in suite discovery information.

##### Parameters

- **prefix** (str) – Repository prefix.
- **format** (string) – Desired return format (json or None).

### Example Request

```
GET http://host/my_prefix/get_optional_keys?format=json HTTP/1.1
```

### Example Response

```
[ "access-list", "description", "endgame_status", "operational_flag", "tag-  
↳list" ]
```

## GET (*str: prefix*) /get\_query\_operators

Returns all the SQL-like operators used to compare column values that you may use in *GET (str:prefix)/query* (page 314) (e.g. eq, ne, contains, like).

#### Parameters

- **prefix** (*str*) – Repository prefix.
- **format** (*string*) – Desired return format (json or None).

### Example Request

```
GET http://host/my_prefix/get_query_operators?format=json HTTP/1.1
```

### Example Response

```
[ "eq", "ge", "gt", "le", "lt", "ne", "contains", "endswith", "ilike", "like  
↳", "match", "startswith" ]
```

## GET (*str: prefix*) /query

Return a list of suites matching all search terms.

#### Parameters

- **prefix** (*str*) – Repository prefix.
- **query** (*list*) – List of queries.
- **format** (*string*) – Desired return format (json or None).
- **all\_revs** (*flag*) – Switch on searching older revisions of current suites and deleted suites.

#### Query Parameters

- **CONJUNCTION** (*str*) – and or or.
- **OPEN\_GROUP** (*str*) – optional, one or more (.
- **FIELD** (*str*) – e.g. idx or description.
- **OPERATOR** (*str*) – e.g. contains or between, one of the operators returned by *GET (str:prefix)/get\_query\_operators* (page 314).
- **VALUE** (*str*) – e.g. euro4m or 200.
- **CLOSE\_GROUP** (*str*) – optional, one or more ).

#### Query Format

```
CONJUNCTION+ [OPEN_GROUP+] FIELD+OPERATOR+VALUE [+CLOSE_GROUP]
```

The first CONJUNCTION is technically superfluous. The OPEN\_GROUP and CLOSE\_GROUP do not have to be used.

Parentheses can be used in the query to group expressions.

**Example Request** Return all current suites that have an `idx` that ends with 78 and also all suites that have the owner bob.

```
GET ↵
→ http://host/my_prefix/query?q=and+idx+endswith+78&q=or+owner+eq+bob&format=json
→ HTTP/1.1
```

**Example Response** Each suite is returned as an entry in a list - each entry is an associative array of property name-value pairs.

```
[{"idx": "mol-aa078", "branch": "trunk", "revision": 200, "owner": "fred",
 "project": "fred's project.", "title": "fred's awesome suite",
 "status": "M ", "access-list": ["fred", "jack"], "description": "awesome
→"},
 {"idx": "mol-aa090", "branch": "trunk", "revision": 350, "owner": "bob",
 "project": "var", "title": "A copy of var.vexcs.", "status": "M ",
 "access-list": ["*"], "operational": "Y"}]
```

#### GET (`str: prefix`) /search

Return a list of suites matching one or more search terms.

##### Parameters

- **prefix** (`str`) – Repository prefix.
- **search** (`list`) – List of queries in the same format as `GET (str:prefix) /query` (page 314)
- **format** (`string`) – Desired return format (json or None).
- **all\_revs** (`flag`) – Switch on searching older revisions of current suites and deleted suites.

**Example Request** Return suites which match var, bob or nowcast.

```
GET http://host/my_prefix/search?var+bob+nowcast&format=json HTTP/1.1
```

**Example Response**

```
[{"idx": "mol-aa090", "branch": "trunk", "revision": 330, "owner": "bob",
 "project": "um", "title": "A copy of um.alpra.", "status": "M ",
 "description": "Bob's UM suite"}, {"idx": "mol-aa092", "branch": "trunk", "revision": 340, "owner": "jim",
 "project": "var", "title": "6D Quantum VAR.", "status": "M ",
 "location": "NAE"}, {"idx": "mol-aa100", "branch": "trunk", "revision": 352, "owner": "ops_
→account",
 "project": "nowcast", "title": "The operational Nowcast suite",
 "status": "M ", "ensemble": "yes"}]
```

## 15.3 Python API

The REST API maps onto the Python `RosieWSClient` back-end which can be used as a standalone Python API.

```
class rosie.ws_client.RosieWSClient(prefixes=None, prompt_func=None, popen=None,
   event_handler=None)
```

A client for the Rosie web service.

##### Parameters

- **prefixes** (*list*) – List of prefix names as strings. (run `rose config rosie-id` for more info).
- **prompt\_func** (*function*) – Optional function for requesting user credentials. Takes and returns the arguments username and password.
- **popen** (`rose.popen.RosePopener`) – Use initiated RosePopener instance create a new one if None.
- **event\_handler** (*object*) – A callable object for reporting popen output, see `rose.reporter.Reporter`.

**address\_lookup** (\*\*kwargs)

Repeat a Rosie query or search by address.

**get\_known\_keys** ()

Return the known query keys.

**get\_optional\_keys** ()

Return the optional query keys.

**get\_query\_operators** ()

Return the query operators.

**hello** (`return_ok_prefixes=False`)

Ask the server to say hello.

**query** (*q*, \*\*kwargs)

Query the Rosie database.

**query\_local\_copies** (`user=None`)

Returns details of the local suites.

As if they had been obtained using a search or query.

**classmethod query\_split** (*args*)

Split a list of arguments into a list of query items.

**search** (*s*, \*\*kwargs)

Search the Rosie database for a matching string.

**set\_prefixes** (*prefixes*)

Replace the default prefixes.

Parameters **prefixes** (*list*) – List of prefix names as strings.

---

CHAPTER  
**SIXTEEN**

---

**OTHER**

- genindex
- *Rose Terms Of Use* (page 317)

## 16.1 Rose Terms Of Use

### 16.1.1 Rose Software: Licence

Rose is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Rose is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Rose. If not, see <http://www.gnu.org/licenses/>.

### 16.1.2 Rose Documentation: Licence

You may use and re-use Crown copyright information from any part of the Rose Documentation (not including logos) free of charge in any format or medium, under the terms and conditions of the Open Government Licence, provided it is reproduced accurately and not used in a misleading context. Where any of the Crown copyright items on this website are being republished or copied to others, the source of the material must be identified and the copyright status acknowledged.

See the [Open Government Licence](http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/) (<http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/>) for the full conditions of this licence and for information on how to attribute the source of material.

We encourage users to establish hypertext links to the Rose Documentation.

### 16.1.3 Rose: Maintenance

The Rose software and Rose Documentation are maintained by the [Modelling Infrastructure Support Systems Team](#) ([metomi@metoffice.gov.uk](mailto:metomi@metoffice.gov.uk)), Met Office, FitzRoy Road, Exeter, Devon, EX1 3PB, UK.



## HTTP ROUTING TABLE

### /(str:prefix)

GET (str:prefix)/get\_known\_keys, 313  
GET (str:prefix)/get\_optional\_keys,  
      313  
GET (str:prefix)/get\_query\_operators,  
      314  
GET (str:prefix)/query, 314  
GET (str:prefix)/search, 315



## PYTHON MODULE INDEX

r

`rose.config`, 228

`rose.macro`, 305



# INDEX

## A

act\_from\_files() (rose.upgrade.MacroUpgrade method), 310  
add() (rose.config.ConfigNode method), 233  
add\_report() (rose.macro.MacroBase method), 306  
add\_setting() (rose.upgrade.MacroUpgrade method), 310  
add\_variable\_widget(), 299  
address\_lookup() (rosie.ws\_client.RosieWSClient method), 316  
AnalysisTask (class in rose.apps.rose\_ana), 151  
application directory, 179

## B

batch system, 179

## C

can\_miss\_opt\_conf\_key() (rose.config.ConfigLoader static method), 230  
change\_setting\_value() (rose.upgrade.MacroUpgrade method), 310  
code (rose.config.ConfigSyntaxError.exc attribute), 242  
col\_num (rose.config.ConfigSyntaxError.exc attribute), 242  
cold start, 179  
config (rose.apps.rose\_ana.AnalysisTask.self attribute), 151  
ConfigDumper (class in rose.config), 229  
ConfigLoader (class in rose.config), 230  
ConfigNode (class in rose.config), 232  
ConfigNodeDiff (class in rose.config), 237  
ConfigSyntaxError, 242  
cycle, 179  
cycle point, 180  
cycling, 180  
Cycl suite, 189

## D

datetime cycling, 180  
delete\_removed() (rose.config.ConfigNodeDiff method), 238  
dependency, 180  
directive, 181  
dump() (in module rose.config), 243  
dump() (rose.config.ConfigDumper method), 230

## E

enable\_setting() (rose.upgrade.MacroUpgrade method), 311  
environment variable  
  NPROC, 275  
  ROSE\_APP\_COMMAND\_KEY, 275  
  ROSE\_APP\_OPT\_CONF\_KEYS, 136, 137, 275  
  ROSE\_BUNCH\_LOG\_PREFIX, 275  
  ROSE\_CONF\_PATH, 275  
  ROSE\_CYCLING\_MODE, 275  
  ROSE\_DATA, 276  
  ROSE\_DATAAC, 122, 123, 276  
  ROSE\_DATAAC????, 276  
  ROSE\_ETC, 276  
  ROSE\_FILE\_INSTALL\_ROOT, 226, 276  
  ROSE\_HOME, 276  
  ROSE\_HOME\_BIN, 276  
  ROSE\_LAUNCHER, 277  
  ROSE\_LAUNCHER\_FILEOPTS, 277  
  ROSE\_LAUNCHER\_LIST, 277  
  ROSE\_LAUNCHER\_POSTOPTS, 277  
  ROSE\_LAUNCHER\_PREOPTS, 277  
  ROSE\_LAUNCHER\_ULIMIT\_OPTS, 277  
  ROSE\_META\_PATH, 206, 216, 277  
  ROSE\_NS, 278  
  ROSE\_ORIG\_HOST, 278  
  ROSE\_SUITE\_DIR, 278, 290  
  ROSE\_SUITE\_DIR\_REL, 278  
  ROSE\_SUITE\_NAME, 278  
  ROSE\_SUITE\_OPT\_CONF\_KEYS, 137, 278  
  ROSE\_TASK\_APP, 110, 278  
  ROSE\_TASK\_CYCLE\_TIME, 278, 289  
  ROSE\_TASK\_LOG\_DIR, 278  
  ROSE\_TASK\_LOG\_ROOT, 278  
  ROSE\_TASK\_MIRROR\_TARGET, 279  
  ROSE\_TASK\_N\_JOBS, 279, 284  
  ROSE\_TASK\_NAME, 279  
  ROSE\_TASK\_OPTIONS, 279, 284  
  ROSE\_TASK\_PREFIX, 279  
  ROSE\_TASK\_SUFFIX, 279  
  ROSE\_UTIL, 280  
  ROSE\_VERSION, 280

## F

family, 181  
family inheritance, 181

family trigger, 181

file\_name (rose.config.ConfigSyntaxError.exc attribute), 242

FileCommandPattern (class in rose.apps.ana\_builtin.grepper), 286

FilePattern (class in rose.apps.ana\_builtin.grepper), 286  
final cycle point, 182

## G

get() (rose.config.ConfigNode method), 233

get\_added() (rose.config.ConfigNodeDiff method), 238  
get\_all\_keys() (rose.config.ConfigNodeDiff method), 238

get\_as\_opt\_config() (rose.config.ConfigNodeDiff method), 239

get\_filter() (rose.config.ConfigNode method), 234

get\_known\_keys() (rosie.ws\_client.RosieWSClient method), 316

get\_metadata\_for\_config\_id() (rose.macro.MacroBase method), 306

get\_modified() (rose.config.ConfigNodeDiff method), 239

get\_optional\_keys() (rosie.ws\_client.RosieWSClient method), 316

get\_query\_operators() (rosie.ws\_client.RosieWSClient method), 316

get\_removed() (rose.config.ConfigNodeDiff method), 239

get\_resource\_path() (rose.macro.MacroBase method), 307

get\_reversed() (rose.config.ConfigNodeDiff method), 240

get\_setting\_value() (rose.upgrade.MacroUpgrade method), 311

get\_value() (rose.config.ConfigNode method), 234

graph, 182

graph string, 182

## H

hello() (rosie.ws\_client.RosieWSClient method), 316

## I

ignore\_setting() (rose.upgrade.MacroUpgrade method), 311

initial cycle point, 182

integer cycling, 182

inter-cycle dependency, 182

is\_ignored() (rose.config.ConfigNode method), 235

ISO8601, 183

ISO8601 datetime, 184

ISO8601 duration, 184

## J

job, 184

job host, 184

job log, 184

job log directory, 184

job script, 184

job submission number, 184

## K

kgo\_db (rose.apps.rose\_ana.AnalysisTask.self attribute), 151

## L

line (rose.config.ConfigSyntaxError.exc attribute), 242  
line\_num (rose.config.ConfigSyntaxError.exc attribute), 242

load() (in module rose.config), 243

load() (rose.config.ConfigLoader method), 230

load\_with\_opts() (rose.config.ConfigLoader method), 231

## M

MacroBase (class in rose.macro), 305

MacroReport (class in rose.macro), 307

MacroUpgrade (class in rose.upgrade), 309

metadata, 185

## P

parameterisation, 185

popen (rose.apps.rose\_ana.AnalysisTask.self attribute), 151

pretty\_format\_config() (rose.macro.MacroBase method), 307

## Q

qualifier, 186

query() (rosie.ws\_client.RosieWSClient method), 316  
query\_local\_copies() (rosie.ws\_client.RosieWSClient method), 316

query\_split() (rosie.ws\_client.RosieWSClient class method), 316

## R

recurrence, 186

reload, 186

reload\_variable\_widget(), 299

remove\_setting() (rose.upgrade.MacroUpgrade method), 311

remove\_variable\_widget(), 299

rename\_setting() (rose.upgrade.MacroUpgrade method), 312

reporter (rose.apps.rose\_ana.AnalysisTask.self attribute), 151

restart, 186

Rose app, 186

Rose application, 186

Rose application configuration, 186

Rose built-in application, 187

Rose configuration, 187

Rose metadata, 185

Rose suite configuration, 187

rose.config (module), 228

rose.macro (module), 305

ROSE\_APP\_OPT\_CONF\_KEYS, 136, 137

ROSE\_DATAAC, 122, 123  
 ROSE\_FILE\_INSTALL\_ROOT, 226  
 ROSE\_LAUNCHER, 277  
 ROSE\_META\_PATH, 206, 216  
 ROSE\_SUITE\_DIR, 290  
 ROSE\_SUITE\_OPT\_CONF\_KEYS, 137  
 ROSE\_TASK\_APP, 110  
 ROSE\_TASK\_CYCLE\_TIME, 289  
 ROSE\_TASK\_N\_JOBS, 284  
 ROSE\_TASK\_OPTIONS, 284  
 Rosie Suite, 187  
 RosieWSClient (class in `rosie.ws_client`), 315  
 run directory, 187

## S

search() (`rosie.ws_client.RosieWSClient` method), 316  
 set() (`rose.config.ConfigNode` method), 235  
 set\_added\_setting() (`rose.config.ConfigNodeDiff` method), 240  
 set\_from\_configs() (`rose.config.ConfigNodeDiff` method), 241  
 set\_modified\_setting() (`rose.config.ConfigNodeDiff` method), 241  
 set\_prefixes() (`rosie.ws_client.RosieWSClient` method), 316  
 set\_removed\_setting() (`rose.config.ConfigNodeDiff` method), 242  
 share directory, 188  
 shutdown, 189  
`SingleCommandPattern` (class in `rose.apps.ana_builtin.grepper`), 287  
`SingleCommandStatus` (class in `rose.apps.ana_builtin.grepper`), 287  
 sort\_element() (in module `rose.config`), 243  
 sort\_settings() (in module `rose.config`), 243  
 stalled state, 188  
 stalled suite, 188  
 standard\_format\_config() (`rose.macro.MacroBase` method), 307  
 start, 188  
 startup, 188  
`STATE_NORMAL` (`rose.config.ConfigNode` attribute), 233  
`STATE_SYS_IGNORED` (`rose.config.ConfigNode` attribute), 233  
`STATE_USER_IGNORED` (`rose.config.ConfigNode` attribute), 233  
 stop, 189  
 suite, 189  
 suite directory, 189  
 suite log, 189  
 suite log directory, 189  
 suite server program, 190

## T

task, 190  
 task state, 190  
 task trigger, 190

## U

unset() (`rose.config.ConfigNode` method), 236  
 update\_ignored(), 299

## W

walk() (`rose.config.ConfigNode` method), 236  
 wall-clock time, 190  
 warm start, 190  
 work directory, 190