gc3pie Documentation

Release 2.3.dev (SVN** $Revision$**)

Sergio Maffioletti, Antonio Messina, Mark Monroe, Riccardo Mu

August 23, 2015

Contents

CHAPTER 1

Introduction

GC3Pie is a Python package for running large job campaigns on diverse batch-oriented execution environments (for instance: a Sun/Oracle/Open Grid Engine cluster, or the Swiss National Distributed Computing Infrastructure SMSCG). It also provides facilities for implementing command-line driver scripts, in the form of Python object classes whose behavior can be customized by overriding specified object methods.

GC3Pie documentation is divided in three sections:

- *User Documentation*: info on how to install, configure and run GC3Pie applications.
- *Programmer Documentation*: info for programmers who want to use the GC3Pie libraries to write their own scripts and applications.
- *Developer Documentation*: detailed information on how to contribute to GC3Pie and get your code included in the main library.

CHAPTER 2

Table of Contents

2.1 User Documentation

This section describes how to install and configure GC3Pie, and how to run *The GC3Apps software* and *The GC3Utils software*.

2.1.1 Table of Contents

Installation of GC3Pie

Quick start

We provide an installation script which automatically tries to install GC3pie in your home directory. The quick installation procedure has only been tested on variants of the GNU/Linux operating system. (However, the script should work on MacOSX as well, provided you follow the preparation steps outlined in the "MacOSX installation" section below.)

To install GC3Pie just type this at your terminal prompt:

```
sh -c "$(wget -O- http://gc3pie.googlecode.com/svn/install.sh)"
```

If *wget* is not installed in your computer, you can use *curl* instead:

```
sh -c "$(curl -s http://gc3pie.googlecode.com/svn/install.sh)"
```

The above command creates a directory `$HOME/gc3pie` and installs the latest release of GC3Pie and all its dependencies into it.

In case you have trouble running the installation script, please send an email to gc3pie@googlegroups.com. (In order to post a message, you first need to subscribe. To post a message without being subscribed, please use the web interface at http://dir.gmane.org/gmane.comp.python.gc3pie.) Include the full output of the script in your email, in order to help us to identify the problem.

Now you can check your GC3Pie installation; follow the on-screen instructions to activate the virtual environment. Then, just type the command:

```
gc3utils --help
```

and you should see the following output appear on your screen:

```
Usage: gc3utils COMMAND [options]

Command `gc3utils` is a unified front-end to computing resources.
You can get more help on a specific sub-command by typing::
  gc3utils COMMAND --help
where command is one of these:
```

```
clean
cloud
get
info
kill
resub
select
servers
stat
tail
```

If you get some errors, do not despair! The GC3Pie users mailing-list is there to help you :-) (You can also post to the same forum using a web interface at http://dir.gmane.org/gmane.comp.python.gc3pie.)

With the default configuration file, GC3Pie is set up to only run jobs on the computer where it is installed. To run jobs on remote resources, you need to edit the configuration file; the ConfigurationFile Wiki page provides an explanation of the syntax.

Non-standard installation options

The installation script accept a few options that select alternatives to the standard behavior. In order to use these options, you have to:

1. download the installation script into a file named `install.sh`:

```
wget http://gc3pie.googlecode.com/svn/install.sh
```

2. run the command:

```
sh ./install.sh [options]
```

   replacing the string `[options]` with the actual options you want to pass to the script.

The accepted options are as follows:

`--feature LIST`

Install optional features (comma-separated list). Currently defined features are:

- `openstack`: support running jobs in VMs on OpenStack clouds
- `ec2`: support running jobs in VMs on OpenStack clouds
- `optimizer`: install math libraries needed by the optimizer library

For instance, to install all features use `-a openstack,ec2,optimizer`. To install no optional feature, use `-a none`.

By default, all cloud-related features are installed.

`-d DIRECTORY`

Install GC3Pie in location `DIRECTORY` instead of `$HOME/gc3pie`

`--overwrite`

Overwrite the destination directory if it already exists. Default behavior is to abort installation.

`--develop`

Instead of installing the latest *release* of GC3Pie, it will install the *development branch* from the SVN repository.

`--yes`

Run non-interactively, and assume a "yes" reply to every question.

`-p PYTHON`

Uses the given `PYTHON` program as python interpreter. By default the installation script looks for a `python` binary in the standard `$PATH`.

`--no-gc3apps`

Do not install any of the GC3Apps, e.g., `gcodeml`, `grosetta` and `ggamess`.

Manual installation

In case you can't or don't want to use the automatic installation script, the following instructions will guide you through all the steps needed to manually install GC3Pie on your computer.

These instructions show how to install GC3Pie from the GC3 source repository into a separate python environment (called virtualenv). Installation into a virtualenv has two distinct advantages:

- All code is confined in a single directory, and can thus be easily replaced/removed.
- Better dependency handling: additional Python packages that GC3Pie depends upon can be installed even if they conflict with system-level packages.

0. Install software prerequisites:
   - On Debian/Ubuntu, install packages: `subversion`, `python-dev`, `python-profiler` and the C/C++ compiler:

```
apt-get install subversion python-dev python-profiler gcc g++
```

   - On CentOS5, install packages `subversion` and `python-devel` and the C/C++ compiler:

```
yum install subversion python-devel gcc gcc-c++
```

   - On other Linux distributions, you will need to install:
     - the `svn` command (from the SubVersion VCS)
     - Python development headers and libraries (for installing extension libraries written in C/C++)
     - the Python package `pstats` (it's part of the Python standard library, but sometimes it needs separate installation)
     - a C/C++ compiler (this is usually installed by default).

1. If virtualenv is not already installed on your system, get the Python package and install it:

```
wget http://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.7.tar.gz
tar -xzf virtualenv-1.7.tar.gz && rm virtualenv-1.7.tar.gz
cd virtualenv-1.7/
```

   If you are installing as *root*, the following command is all you need:

```
python setup.py install
```

   If instead you are installing as a normal, unprivileged user, things get more complicated:

```
export PYTHONPATH=$HOME/lib64/python:$HOME/lib/python:$PYTHONPATH
export PATH=$PATH:$HOME/bin
mkdir -p $HOME/lib/python
python setup.py install --home $HOME
```

   You will also *have to* add the two *export* lines above to the:

   - *$HOME/.bashrc* file, if using the *bash* shell or to the
   - *$HOME/.cshrc* file, if using the *tcsh* shell.

   In any case, once *virtualenv* has been installed, you can exit its directory and remove it:

```
cd ..
rm -rf virtualenv-1.7
```

2. Create a virtualenv to host the GC3Pie installation, and `cd` into it:

```
virtualenv --system-site-packages $HOME/gc3pie
cd $HOME/gc3pie/
source bin/activate
```

   In this step and in the following ones, the directory `$HOME/gc3pie` is going to be the installation folder of GC3Pie. You can change this to another directory path; any directory that's writable by your Linux account will be OK.

   If you are installing system-wide as `root`, we suggest you install GC3Pie into `/opt/gc3pie` instead.

3. Check-out the `gc3pie` files in a `src/` directory:

```
svn co http://gc3pie.googlecode.com/svn/branches/2.0/gc3pie src
```

4. Install the `gc3pie` in "develop" mode, so any modification pulled from subversion is immediately reflected in the running environment:

```
cd src/
env CC=gcc ./setup.py develop
cd .. # back into the `gc3pie` directory
```

   This will place all the GC3Pie command into the `gc3pie/bin/` directory.

5. GC3Pie comes with driver scripts to run and manage large families of jobs from a few selected applications. These scripts are not installed by default because not everyone needs them.

   Run the following commands to install the driver scripts for the applications you need:

```
# if you are insterested in GAMESS, do the following
ln -s '../src/gc3apps/gamess/ggamess.py' bin/ggamess

# if you are insterested in Rosetta, do the following
ln -s '../src/gc3apps/rosetta/gdocking.py' bin/gdocking
ln -s '../src/gc3apps/rosetta/grosetta.py' bin/grosetta

# if you are insterested in Codeml, do the following
ln -s '../src/gc3apps/codeml/gcodeml.py' bin/gcodeml
```

6. Now you can check your GC3Pie installation; just type the command:

```
gc3utils --help
```

   and you should see the following output appear on your screen:

```
Usage: gc3utils COMMAND [options]

Command `gc3utils` is a unified front-end to computing resources.
You can get more help on a specific sub-command by typing::
  gc3utils COMMAND --help
where command is one of these:
  clean
  cloud
  get
  info
  kill
  resub
  select
  servers
  stat
  tail
```

If you get some errors, do not despair! The *GC3Pie users mailing-list <gc3pie@googlegroups.com>* is there to help you :-) (You can also post to the same forum using the web interface at http://dir.gmane.org/gmane.comp.python.gc3pie.)

7. With the default configuration file, GC3Pie is set up to only run jobs on the computer where it is installed. To run jobs on remote resources, you need to edit the configuration file; the ConfigurationFile Wiki page provides an explanation of the syntax.

Upgrade

If you used the installation script, the fastest way to upgrade is just to reinstall:

0. De-activate the current GC3Pie virtual environment:

```
deactivate
```

(If you get an error "command not found", do not worry and proceed on to the next step; in case of other errors please stop here and report to the *GC3Pie users mailing-list <mailto:gc3pie.googlegroups.com>*.)

1. Move the `$HOME/gc3pie` directory to another location, e.g.:

```
mv $HOME/gc3pie $HOME/gc3pie.OLD
```

2. Reinstall GC3Pie using the quick-install script (top of this page).
3. Once you have verified that your new installation is working, you can remove the `$HOME/gc3pie.OLD` directory.

If instead you installed GC3Pie using the "manual installation" instructions, then the following steps will update GC3Pie to the latest version in the code repository:

1. **cd** to the directory containing the GC3Pie virtualenv; assuming it is named `gc3pie` as in the above installation instructions, you can issue the commands:

```
cd $HOME/gc3pie # use '/opt/gc3pie' if root
```

2. Activate the virtualenv:

```
source bin/activate
```

3. Upgrade the *gc3pie* source and run the `setup.py` script again:

```
cd src
svn up
env CC=gcc ./setup.py develop
```

Note: A major restructuring of the SVN repository took place in r1124 to r1126 (Feb. 15, 2011); if your sources are older than SVN r1124, these upgrade instructions will not work, and you must *reinstall completely*. You can check what version the SVN sources are, by running the *svn info* command in the *src* directory: watch out for the *Revision:* line.

MacOSX Installation

Installation on MacOSX machines is possible, however there are still a few issues. If you need MacOSX support, please let us know on the *GC3Pie users mailing-list <mailto:gc3pie@googlegroups.com>* or by posting a message using the web interface at http://dir.gmane.org/gmane.comp.python.gc3pie.

1. Standard usage of the installation script (i.e., with no options) works, but you have to use *curl* since *wget* is not installed by default.
2. In order to install GC3Pie you will need to install XCode and, in some of the MacOSX versions, also the *Command Line Tools for XCode*

3. Options can only be given in the abbreviated one-letter form (e.g., `-d`); the long form (e.g., `--directory`) will not work.
4. The *shellcmd* backend of GC3Pie depends on the GNU `time` command, which is not installed on MacOSX by default. This means that with a standard MacOSX installation the *shellcmd* resource will **not** work. However:
   - other resources, like *pbs* via *ssh* transport, will work.
   - you can install the GNU time command either via MacPorts, Fink, Homebrew or from the this url. After installing it you don't need to update your `PATH` environment variable, it's enough to set the `time_cmd` option in your GC3Pie configuration file.

HTML Documentation

HTML documentation for the GC3Libs programming interface can be read online at:

http://gc3pie.googlecode.com/svn/branches/2.0/gc3pie/docs/html/index.html

If you installed GC3Pie manually, or if you installed it using the `install.sh` script with the `--develop` option, you can also access a local copy of the documentation from the sources:

```
cd $HOME/gc3pie # or wherever the gc3pie virtualenv is installed
cd src/docs
make html
```

Note that you need the Python package Sphinx in order to build the documentation locally.

Configuration File

Location

All commands in *The GC3Apps software* and *The GC3Utils software* read two configuration files at startup:

- system-wide one located at :file:`/etc/gc3/gc3pie.conf`, and
- a user-private one at :file:`~/.gc3/gc3pie.conf`.

Both files are optional, but at least one of them must exist.

Both files use the same format. The system-wide one is read first, so that users can override the system-level configuration in their private file. Configuration data from corresponding sections in the two configuration files is merged; the value in the user-private file overrides the one from the system-wide configuration.

If you try to start any GC3Utils command without having a configuration file, a sample one will be copied to the user-private location :file:`~/.gc3/gc3pie.conf` and an error message will be displayed, directing you to edit the sample file before retrying.

Configuration file format

The GC3Pie configuration file follows the format understood by Python ConfigParser, which is very close to the syntax used in MS-Windows `.INI` files. See http://docs.python.org/library/configparser.html for reference.

The GC3Libs configuration file consists of several configuration blocks. Each configuration block (section) starts with a keyword in square brackets and contains the configuration options for a specific part.

The following sections are used by the GC3Apps/GC3Utils programs:

- `[DEFAULT]` – this is for global settings.
- `[auth/`*`name`*`]` – these are for settings related to identity/authentication (identifying yourself to clusters & grids).
- `[resource/`*`name`*`]` – these are for settings related to a specific computing resource (cluster, grid, etc.)

Sections with other names are allowed but will be ignored.

The `DEFAULT` section

The `[DEFAULT]` section is optional.

Values defined in the `[DEFAULT]` section can be used to insert values in other sections, using the `%(name)s` syntax. See documentation of the Python SafeConfigParser object at http://docs.python.org/library/configparser.html for an example.

`auth` sections

There can be more than one `[auth]` section.

Each authentication section must begin with a line of the form:

```
[auth/name]
```

where the *name* portion is any alphanumeric string.

You can have as many `[auth/name]` sections as you want; any name is allowed provided it's composed only of letters, numbers and the underscore character _.

This allows you to define different auth methods for different resources. Each `[resource/name]` section will reference one (and one only) authentication section.

Authentication types Each `auth` section *must* specify a `type` setting.

`type` defines the authentication type that will be used to access a resource. There are three supported authentication types:

- `ssh`; use this for resources that will be accessed by opening an SSH connection to the front-end node of a cluster.
- `voms-proxy`: uses `voms-proxy-init` to generate a proxy; use for resources that require a VOMS-enabled Grid proxy.
- `grid-proxy`: uses `grid-proxy-init` to generate a proxy; use for resources that require a Grid proxy (but no VOMS extensions).
- `ec2`: use this for a EC2-compatible cloud resource.

For the `ssh`-type auth, the following keys must be provided:

- `type`: must be `ssh`
- `username`: must be the username to log in as on the remote machine

The following configuration keys are instead optional:

- `port`: TCP port number where the SSH server is listening. The default value 22 is fine for almost all cases; change it only if you know what you are doing.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from (default: `$HOME/ssh/config`:file:. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

For the `ec2`-type auth, the following keys *can* be provided. If they are not found, the value of the corresponding environment variable will be used instead, if found, otherwise an error will be raised.

- `ec2_access_key`: Your personal access key to authenticate against the specific cloud endpoint. If not found, the environment variable `EC2_ACCESS_KEY` will be used.
- `ec2_secret_key`: Your personal secret key associated with the above `ec2_access_key`. If not found, the environment variable `EC2_SECRET_KEY` will be used.

Any other key/value pair will be ignored.

For the `voms-proxy` type auth, the following keys must be provided:

- `type`: must be `voms-proxy`
- `vo`: the VO to authenticate with (passed directly to `voms-proxy-init` as argument to the `--vo` command-line switch)
- `cert_renewal_method`: see below.
- `remember_password`: see below.

Any other key/value pair will be ignored.

For the `grid-proxy` type auth, the following keys must be provided:

- `type`: must be `grid-proxy`
- `cert_renewal_method`: see below.
- `remember_password`: see below.

Any other key/value pair will be ignored.

For the `voms-proxy` and `grid-proxy` authentication types, the `cert_renewal_method` setting specifies whether GC3Libs should attempt to get a certificate if the current one is expired or otherwise invalid. Currently there are two supported `cert_renewal_method` types:

- `slcs`: user certificate is generated through an invocation of the `slcs-init`:command: program.
- `manual`: user certificate is generated/renewed though an external process and has to be performed by the user outside of the scope of GC3Pie. In this case, if the user certificate is expired, invalid or non-existent, GC3Pie will fail to authenticate.

For the `slcs` certificate renewal method, the following keys must be provided:

- `aai_username`: passed directly to **`slcs-init`** as argument to the `--user` command-line switch.
- `idp`: passed directly to **`slcs-init`** as argument to the `--idp` command-line switch.

For the `manual` certificate renewal method, no additional keys are required.

The `remember_password` entry (optional) must be set to a boolean value (the strings `1`, ``yes``, `true` and `on` are interpreted as boolean "true"; any other value counts as "false"). If set to a true value, the `remember_password` entry instructs GC3Pie to keep the password used for this authentication in the program's main memory; this implies that you will be asked for the password at most once per program invocation. This setting is optional, and defaults to "false". Keeping passwords in memory is bad security practice; do not set this option to "true" unless you understand the implications.

Example 1. The following example `auth` section shows how to configure GC3Pie for using SWITCHaai SLCS services to generate a certificate and a VOMS proxy to access the Swiss National Distributed Computing Infrastructure SMSCG:

```
[auth/smscg]
type = voms-proxy
cert_renewal_method = slcs
aai_username = <aai_user_name> # SWITCHaai/Shibboleth user name
idp= uzh.ch
vo = smscg
```

Example 2. The following configuration sections are used to set up two different accounts, that GC3Pie programs can use. Which account should be used on which computational resource is defined in the *resource sections* (see below).

```
[auth/ssh1]
type = ssh
username = murri # your username here

[auth/ssh2] # I use a different account name on some resources
type = ssh
username = rmurri
# read additional options from this SSH config file
ssh_config = ~/.ssh/alt-config
```

Example 3. The following configuration section is used to access an EC2 resource (access and secret keys are of course invalid :)):

```
[auth/hobbes]
type=ec2
ec2_access_key=1234567890qwertyuiopasdfghjklzxc
ec2_secret_key=cxzlkjhgfdsapoiuytrewq0987654321
```

`resource` sections

Each resource section must begin with a line of the form:

`[resource/`*`name`*`]`

You can have as many `[resource/`*`name`*`]` sections as you want; this allows you to define many different resources. Each `[resource/`*`name`*`]` section must reference one (and one only) `[auth/`*`name`*`]` section (by its `auth` key).

Resources currently come in several flavours, distinguished by the value of the `type` key:

- If `type` is `sge`, then the resource is a Grid Engine batch system, to be accessed by an SSH connection to its front-end node.
- If `type` is `pbs`, then the resource is a Torque/PBS batch system, to be accessed by an SSH connection to its front-end node.
- If `type` is `lsf`, then the resource is a LSF batch system, to be accessed by an SSH connection to its front-end node.
- If `type` is `slurm`, then the resource is a SLURM batch system, to be accessed by an SSH connection to its front-end node.
- If `type` is `shellcmd`, then the resource is the computer where the GC3Pie script is running and applications are executed by just spawning a local UNIX process.
- If `type` is `ec2+shellcmd`, then the resource is a cloud with EC2-compatible APIs, and applications are run on Virtual Machines spawned on the cloud.

All `[resource/`*`name`*`]` sections (except those of `shellcmd` type) *must* reference a valid `auth/`$\star\star\star$ section. Resources of `sge`, `pbs`, `lsf` and `slurm` type can only reference :command:`ssh` type sections.

Some configuration keys are commmon to all resource types:

- `type`: Resource type, see above.
- `auth`: the name of a valid `[auth/`*`name`*`]` section; only the authentication section name (after the `/`) must be specified.
- `max_cores_per_job`: Maximum number of CPU cores that a job can request; a resource will be dropped during the brokering process if a job requests more cores than this.
- `max_memory_per_core`: Max amount of memory (expressed in GBs) that a job can request.

- `max_walltime`: Maximum job running time (in hours).
- `max_cores`: Total number of cores provided by the resource.
- `architecture`: Processor architecture. Should be one of the strings `x86_64` (for 64-bit Intel/AMD/VIA processors), `i686` (for 32-bit Intel/AMD/VIA x86 processors), or `x86_64,i686` if both architectures are available on the resource.
- `time_cmd`: Used only when `type` is `shellcmd`. The *time* program is used as wrapper for the application in order to collect informations about the execution when running without a real *LRMS*.
- **`prologue`: Used only when `type` is `pbs`, `lsf`,** `slurm` or `sge`. The content of the *prologue* script will be *inserted* into the submission script and it's executed before the real application. It is intended to execute some shell commands needed to setup the execution environment before running the application (e.g. running a *module load ...* command). The script **must** be a valid, plain */bin/sh* script.
- `<application_name>_prologue`: Same as `prologue`, but it is used only when `<application_name>` matches the name of the application. Valid application names are: *zods*, *gamess*, *turbomole*, *codeml*, *rosetta*, *rosetta_docking*, *geotop*. If both `prologue` and `<application_name>_prologue` options are defined, the content of both files is included in the submission script (first `prologue`, then `<application_name>_prologue`).
- **`prologue_content`: Used only when `type` is `pbs`, `lsf`,** `slurm` or `sge`. A (possibly multi-line) string that will be *inserted* into the submission script and executed before the real application. Its value will be inserted after *any* other `prologue`, `<application_name>_prologue` option, if present.
- `<application_name>_prologue_content`: Same as `prologue_content`, but it is used only when `<application_name>` matches the name of the application. Valid application names are: *zods*, *gamess*, *turbomole*, *codeml*, *rosetta*, *rosetta_docking*, *geotop*. Its value will be inserted after *any* other `prologue`, `<application_name>_prologue`, `prologue_content` option, if present.
- **`epilogue`: Used only when `type` is `pbs`, `lsf`,** `slurm` or `sge`. The content of the *epilogue* script will be *inserted* into the submission script and it's executed after the real application. The script **must** be a valid, plain */bin/sh* script.
- `<application_name>_epilogue`: Same as `epilogue`, but it is used only when `<application_name>` matches the name of the application. Valid application names are: *zods*, *gamess*, *turbomole*, *codeml*, *rosetta*, *rosetta_docking*, *geotop*. If both `epilogue` and `<application_name>_epilogue` options are defined, the content of both files is included in the submission script (first `epilogue`, then `<application_name>_epilogue`).
- **`epilogue_content`: Used only when `type` is `pbs`, `lsf`,** `slurm` or `sge`. A (possibly multi-line) string that will be *inserted* into the submission script and executed after the real application. Its value will be inserted after *any* other `epilogue`, `<application_name>_epilogue` option, if present.
- `<application_name>_epilogue_content`: Same as `epilogue_content`, but it is used only when `<application_name>` matches the name of the application. Valid application names are: *zods*, *gamess*, *turbomole*, *codeml*, *rosetta*, *rosetta_docking*, *geotop*. Its value will be inserted after *any* other `epilogue`, `<application_name>_epilogue`, `epilogue_content` option, if present.

`sge` resources The following configuration keys are required in a `sge`-type resource section:

- `frontend`: should contain the FQDN (Fully-qualified domain name) of the SGE front-end node. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.
- `transport`: Possible values are: `ssh` or `local`. If `ssh`, we try to connect to the host specified in `frontend` via SSH in order to execute SGE commands. If `local`, the SGE commands are run directly on the machine where GC3Pie is installed.

To submit parallel jobs to SGE, a "parallel environment" name must be specified. You can specify the PE to be used with a specific application using a configuration parameter *application name* + `_pe` (e.g., `gamess_pe`, `zods_pe`); the `default_pe` parameter dictates the parallel environment to use if no application-specific one

is defined. *If neither the application-specific, nor the ''default_pe'' parallel environments are defined, then it will not be possible to submit parallel jobs.*

When a job has finished, the SGE batch system does not (by default) immediately write its information into the accounting database. This creates a time window during which no information is reported about the job by SGE, as if it never existed. In order not to mistake this for a "job lost" error, GC3Libs allow a "grace time": **qacct** job information lookups are allowed to fail for a certain time span after the first time **qstat** failed. The duration of this time span is set with the `sge_accounting_delay` parameter, whose default is 15 seconds (matches the default in SGE, as of release 6.2):

- `sge_accounting_delay`: Time (in seconds) a failure in **qacct** will *not* be considered critical.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the SGE backend:

- `qsub`: submit a job.
- `qacct`: get info on resources used by a job.
- `qdel`: cancel a job.
- `qstat`: get the status of a job or the status of available resources.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

---

pbs resources The following configuration keys are required in a `pbs`-type resource section:

- `transport`: Possible values are: `ssh` or `local`. If `ssh`, we try to connect to the host specified in `frontend` via SSH in order to execute Troque/PBS commands. If `local`, the Torque/PBS commands are run directly on the machine where GC3Pie is installed.
- `frontend`: should contain the FQDN of the Torque/PBS front-end node. This configuration item is only relevant if `transport` is `local`. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the PBS backend:

- `queue`: the name of the queue to which jobs are submitted. If empty (the default), no job will be specified during submission.
- `qsub`: submit a job.
- `qdel`: cancel a job.
- `qstat`: get the status of a job or the status of available resources.
- `tracejob`: get info on resources used by a job.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`lsf` resources The following configuration keys are required in a `lsf`-type resource section:

- `transport`: Possible values are: `ssh` or `local`. If `ssh`, we try to connect to the host specified in `frontend` via SSH in order to execute LSF commands. If `local`, the LSF commands are run directly on the machine where GC3Pie is installed.
- `frontend`: should contain the FQDN of the LSF front-end node. This configuration item is only relevant if `transport` is `local`. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the LSF backend:

- `bsub`: submit a job.
- `bjobs`: get the status and resource usage of a job.
- `bkill`: cancel a job.
- `lshosts`: get info on available resources.

LSF commands use a weird formatting: lines longer than 79 characters are wrapped around, and the continuation line starts with a long run of spaces. The length of this run of whitespace seems to vary with LSF version; GC3Pie is normally able to auto-detect it, but there can be a few unlikely cases where it cannot. If this ever happens, the following configuration option is here to help:

- `lsf_continuation_line_prefix_length`: length (in characters) of the whitespace prefix of continuation lines in `bjobs` output. This setting is normally not needed.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`slurm` resources The following configuration keys are required in a `slurm`-type resource section:

- `transport`: Possible values are: `ssh` or `local`. If `ssh`, we try to connect to the host specified in `frontend` via SSH in order to execute SLURM commands. If `local`, the SLURM commands are run directly on the machine where GC3Pie is installed.

- `frontend`: should contain the FQDN of the SLURM front-end node. This configuration item is only relevant if `transport` is `local`. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the SLURM backend:

- `sbatch`: submit a job.
- `scancel`: cancel a job.
- `squeue`: get the status of a job or of the available resources.
- `sacct`: get info on resources used by a job.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`shellcmd` resources The following optional configuration keys are available in a `shellcmd`-type resource section:

- `transport`: Like any other resources, possible values are `ssh` or `local`. Default value is `local`.
- `frontend`: If *transport* is *ssh*, then *frontend* is the FQDN of the remote machine where the jobs will be executed.
- `time_cmd`: `ShellcmdLrms` needs the GNU implementation of the command *time* in order to get resource usage of the submitted jobs. `time_cmd` must contains the path to the binary file if this is different from the standard (`/usr/bin/time`).
- `override`: `ShellcmdLrms` by default will try to gather information on the system the resource is running on, including the number of cores and the available memory. These values may be different from the values stored in the configuration file. If `override` is *True*, then the values automatically discovered will be used. If `override` is *False*, the values in the configuration file will be used regardless of the real values discovered by the resource.
- `spooldir`: Path to a filesystem location where to create temporary working directories for processes executed through this backend. The default value *None* means to use `$TMPDIR` or `/tmp` (see *tempfile.mkftemp* for details).

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`ec2+shellcmd` resource The following configuration options are available for a resource of type `ec2+shellcmd`. If these options are omitted, then the default of the boto python library will be used, which at the time of writing means use the default region on Amazon.

- **`ec2_url`: The URL of the EC2 frontend. On a typical OpenStack** installation this will look like: `https://cloud.gc3.uzh.ch:8773/services/Cloud`, while for amazon it's something like `https://ec2.us-east-1.amazonaws.com` (this is valid for the zone `us-east-1` of course). If no value is specified, the environment variable `EC2_URL` will be used, and if not found an error is raised.
- `ec2_region`: the region you want to access to. Most OpenStack installations only have one region called `nova`.
- `keypair_name`: the name of the keypair to use when creating a new instance on the cloud. If it's not found, a new keypair with this name and the key stored in `public_key` will be used. Please note that if the keypair exists already on the cloud but the associated public key is different from the one stored in `public_key`, then an error is raised and the resource will not be used.
- `public_key`: public key to use when creating the keypair. Please note that GC3Pie will assume that the corresponding private key is stored on a file with the same path but without the `.pub` extension. This private key is necessary in order to access the virtual machines created on the cloud. **Amazon users**: Please note that Amazon does not accept **DSA** keys; use RSA keys only for Amazon resources.
- `vm_auth`: the name of a valid `auth` stanza used to connect to the virtual machine.
- `instance_type`: the instance type (aka *flavor*, aka *size*) you want to use for your virtual machines by default.
- `<application>_instance_type`: you can override the default instance type for a specific application by defining an entry in the configuration file for that application. For example:

```
instance_type=m1.tiny
gc_gps_instance_type=m1.large
```

  will use instance type `m1.large` for the `gc_gps` GC3Pie application, and `m1.tiny` for all the other applications.

- `image_id`: the **ami-id** of the image you want to use. OpenStack users: please note that the ID you will find on the web interface **is not** the *ami-id*. To get the *ami-id* of an image you have to use the command `euca-describe-images` from the `euca2ools` package.

  For Hobbes users: all virtual machines distributed by the GC3 team are in this list of appliances with the corresponding ami-id.

- `<application>_image_id`: you can override the default image id for a specific application by defining an entry in the configuration file for that specific application. For example:

```
image_id=ami-00000048
gc_gps_image_id=ami-0000002a
```

  will use the image `ami-0000002a` for `gc_gps` and image `ami-00000048` for all other applications.

- `security_group_name`: the name of the security group to use. If not found, it will be created using the rules found in `security_group_rules`. If the security group is found but some of the rules in `security_group_rules` are not present, they will be added to the security groups. Please note that if the security group defines some rule which is not listed in `security_group_rules` it will **not** be removed from the security group.
- `security_group_rules`: comma separated list of security rules the `security_group` must have. Each rule is in the form:

```
PROTOCOL:PORT_RANGE_START:PORT_RANGE_END:IP_NETWORK
```

where:

- `PROTOCOL` can be one of `tcp`, `udp`, `icmp`
- `PORT_RANGE_START` and `PORT_RANGE_END` are integers and define the range of ports to allow. If `PROTOCOL` is `icmp` please use `-1` for both values since in `icmp` there is no concept of *port*.
- `IP_NETWORK` is a range of IP to allow in the form `A.B.C.D/N`.

For instance, to allow access to the virtual machine from *any* machine in the internet you can use:

```
tcp:22:22:0.0.0.0/0
```

Please note that in order to be able to access the created virtual machines GC3Pie **needs** to be able to connect via ssh, so the above rule is probably necessary in any gc3pie configuration. (of course, you can allow only your IP address or the IPs of your institution)

- `vm_pool_max_size`: the maximum number of Virtual Machine GC3Pie will start on this cloud. If *0*, there is no predefined limit to the number of virtual machines GC3Pie will spawn.
- `user_data`: the *content* of a script that will run after the startup of the machine. For instance, to automatically upgrade a ubuntu machine after startup you can use:

```
user_data=#!/bin/bash
  aptitude -y update
  aptitude -y safe-upgrade
```

Please note that if you need to span over multiple lines you have to indent the lines after `user_data`, as any indented line in a configuration file is interpreted as a continuation of the previous line.

- `<application>_user_data`: you can override the default userdata for a specific application by defining an entry in the configuration file for that specific application. For example:

```
# user_data=
warholize_user_data = #!/bin/bash
  aptitude update && aptitude -y install imagemagick
```

will install *imagemagick* only for the *warholize* application.

Example `resource` sections *Example 1.* This configuration stanza defines a resource to submit jobs to the Grid Engine cluster whose front-end host is `ocikbpra.uzh.ch`:

```
[resource/ocikbpra]
# A single SGE cluster, accessed by SSH'ing to the front-end node
type = sge
auth = <auth_name> # pick an ``ssh`` type auth, e.g., "ssh1"
transport = ssh
frontend = ocikbpra.uzh.ch
gamess_location = /share/apps/gamess
max_cores_per_job = 80
max_memory_per_core = 2
max_walltime = 2
ncores = 80
```

Example 2. This configuration stanza defines a resource to submit jobs on virtual machines that will be automatically started by GC3Pie on Hobbes, the private OpenStack cloud of the University of Zurich:

```
[resource/hobbes]
enabled=yes
type=ec2+shellcmd
ec2_url=http://hobbes.gc3.uzh.ch:8773/services/Cloud
ec2_region=nova
```

```
auth=ec2hobbes
# These values my be overwritten by the remote resource
max_cores_per_job = 8
max_memory_per_core = 2
max_walltime = 8
max_cores = 32
architecture = x86_64

keypair_name=my_name
# If keypair does not exists, a new one will be created starting from
# `public_key`. Note that if the desired keypair exists, a check is
# done on its fingerprint and a warning is issued if it does not match
# with the one in `public_key`
public_key=~/.ssh/id_dsa.pub
vm_auth=gc3user_ssh
instance_type=m1.tiny
warholize_instance_type = m1.small
image_id=ami-00000048
warholize_image_id=ami-00000035
security_group_name=gc3pie_ssh
security_group_rules=tcp:22:22:0.0.0.0/0, icmp:-1:-1:0.0.0.0/0
vm_pool_max_size = 8
user_data=
warholize_user_data = #!/bin/bash
    aptitude update && aptitude install -u imagemagick
```

Enabling/disabling selected resources

Any resource can be disabled by adding a line `enabled = false` to its configuration stanza. Conversely, a line `enabled = true` will undo the effect of an `enabled = false` line (possibly found in a different configuration file).

This way, resources can be temporarily disabled (e.g., the cluster is down for maintenance) without having to remove them from the configuration file.

You can selectively disable or enable resources that are defined in the system-wide configuration file. Two main use cases are supported: the system-wide configuration file :file:`/etc/gc3/gc3pie.conf` lists and enables all available resources, and users can turn them off in their private configuration file :file:`~/.gc3/gc3pie.conf`; or the system-wide configuration can list all available resources but keep them disabled, and users can enable those they prefer in the private configuration file.

Environment Variables

The following environmental variables affect GC3Pie operations.

GC3PIE_CONF

> Path to an additional configuration file, that is read upon initialization of GC3Pie. If undefined or empty, no additional configuration file is loaded.

GC3PIE_ID_FILE

> Path to the a shared state file, used for recording the "next available" job ID number. By default, it is located at `~/.gc3/next_id.txt`:file:.

GC3PIE_NO_CATCH_ERRORS

> Comma-separated list of unexpected/generic error patterns upon which GC3Pie will not act (by default, ignoring them). Each of these "unignored" errors will be propagated all the way up to top-level. This facilitates running GC3Pie scripts in a debugger and inspecting the code when some unexpected error condition happens.

You can specify which errors to "unignore" by:

- Error class name (e.g., `InputFileError`). Note that this must be the *exact* class name of the error: GC3Pie will not walk the error class hierarchy for matches.
- Function/class/module name: all errors handled in the specified function/class/module will be propagated to the caller.
- Additional keywords describing the error. Please have a look at the source code for these keywords.

GC3PIE_RESOURCE_INIT_ERRORS_ARE_FATAL

If this environmental variable is set to `yes` or `1`, GC3Pie will abort operations immediately if a configured resource cannot be initialized. The default behavior is instead to ignore initialization errors and only abort if *no* resources can be initialized.

The GC3Apps software

GC3Apps is a collection command line front-end to manage submission of a (potentially) large number of computational *job* to different batch processing systems. For example, the GC3Apps commands **`ggamess`** can run GAMESS jobs on the SMSCG infrastructure and on any computational cluster you can `ssh`:command: into.

This chapter is a tutorial for the GC3Apps command-line scripts: it explains the common concepts and features, then goes on to describe the specifics of each command in larger detail.

All GC3Apps scripts share a common set of functionalities, which are derive from a common blueprint, named a *session-based script*, described in Section *Introduction to session-based scripts* below. Script-specific sections detail the scope and options that are unique to a given script.

If you find a technical term whose meaning is not clear to you, please look it up in the *Glossary*. (But feel free to ask on the GC3Pie mailing list if it's still unclear!)

Introduction to *session-based* scripts

All GC3Apps scripts derive their core functionality from a common blueprint, named a *session-based script*. The purpose of this section is to describe this common functionality; script-specific sections detail the scope and options that are unique to a given script. Readers interested in Python programming can find the complete documentation about the *session-based script API* in the `SessionBasedScript` section.

The functioning of GC3Apps scripts revolves around a so-called *session*. A *session* is just a named collection of jobs. For instance, you could group into a single session jobs that analyze a set of related files.

Each time it is run, a GC3Apps script performs the following steps:

1. Reads the session directory and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.
2. Scans the command-line input arguments: if existing jobs do not suffice to analyze the input data, new jobs are added to the session.
3. The status of all existing jobs is updated, output from finished jobs is collected, and new jobs are submitted.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option below.)
4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   Execution can be interrupted at any time by pressing `Ctrl+C`.

Basic command-line usage and options The exact command-line usage of *session-based scripts* varies from one script to the other, so please consult the documentation page for your application. There are quite a number of common options and behaviors, however, which are described here.

Continuous execution While single-pass execution of a GC3Apps script is possible (and sometimes used), it is much more common to keep the script running and let it manage jobs until all are finished. This is accomplished with the following command-line option:

-C NUM, --continuous NUM Keep running, monitoring jobs and possibly submitting new ones or fetching results every NUM seconds.

When all jobs are finished, a GC3Apps script exits even if the `-C` option is given.

Verbose listing of jobs Only a summary of job states is printed by default at the end of step 3., together with the count of jobs that are in the specified state. Use the `-l` option (see below) to get a detailed listing of all jobs.

-l STATE, --state STATE Print a table of jobs including their status.

The *STATE* argument restricts output to jobs in that particular state. It can be a single *state* word (e.g., `RUNNING`) or a comma-separated list thereof (e.g., `NEW,SUBMITTED,RUNNING`).

The pseudo-states `ok` and `failed` are also allowed for selecting jobs in `TERMINATED` state with exit code (respectively) 0 or nonzero.

If *STATE* is omitted, no restriction is placed on job states, and a table of *all* jobs is printed.

Maximum number of concurrent jobs There is a maximum number of jobs that can be in `SUBMITTED` or `RUNNING` state at a given time. GC3Apps scripts will delay submission of newly-created jobs so that this limit is never exceeded. The default limit is 50, but it can be changed with the following command-line option:

-J NUM, --max-running NUM Set the maximum NUMber of jobs (default: 50) in `SUBMITTED` or `RUNNING` state.

Location of output files By default, output files are placed in the same directory where the corresponding input file resides. This can be changed with the following option; it is also possible to specify output locations that vary depending on certain job features.

-o DIRECTORY, --output DIRECTORY Output files from all jobs will be collected in the specified *DIRECTORY* path. If the destination directory does not exist, it is created.

Job control options These command-line options control the requirements and constraints of *new jobs*. Indeed, note that changing the arguments to these options *does not* change the corresponding requirements on jobs that already exist in the session.

-c NUM, --cpu-cores NUM Set the number of CPU cores required for each job (default: 1). *NUM* must be a whole number.

-m GIGABYTES, --memory-per-core GIGABYTES Set the amount of memory required per execution core; (Default: 2GB). Specify this as an integral number followed by a unit, e.g. '512MB' or '4GB'. Valid unit names are: 'B', 'GB', 'GiB', 'KiB', 'MB', 'MiB', 'PB', 'PiB', 'TB', 'TiB', 'kB'.

-r NAME, --resource NAME Submit jobs to a specific *resource*. `NAME` is a reource name or comma-separated list of resource names. Use the command *gservers* to list available resources.

-w DURATION, --wall-clock-time DURATION Set the time limit for each job; default is '8 hours'. Jobs exceeding this limit will be stopped and considered as 'failed'. The duration can be expressed as a whole

number followed by a time unit, e.g., '3600 s', '60 minutes', '8 hours', or a combination thereof, e.g., '2hours 30minutes'. Valid unit names are: 'd', 'day', 'days', 'h', 'hour', 'hours', 'hr', 'hrs', 'm', 'microsec', 'microseconds', 'min', 'mins', 'minute', 'minutes', 'ms', 'nanosec', 'nanoseconds', 'ns', 's', 'sec', 'second', 'seconds', 'secs'.

Session control options This set of options control the placement and contents of the *session*.

-s PATH, --session PATH Store the session information in the directory at *PATH*. (By default, this is a subdirectory of the current directory, named after the script you are executing.)

If *PATH* is an existing directory, it will be used for storing job information, and an index file (with suffix `.csv`) will be created in it. Otherwise, the job information will be stored in a directory named after *PATH* with a suffix `.jobs` appended, and the index file will be named after *PATH* with a suffix `.csv` added.

-N, --new-session Discard any information saved in the session directory (see the `--session` option) and start a new session afresh. Any information about jobs previously recorded in the session is lost.

-u, --store-url URL Store GC3Pie job information at the persistent storage specified by *URL*. The *URL* can be any form that is understood by the *gc3libs.persistence.make_store()* function (which see for details). A few examples:

- `sqlite` – the jobs are stored in a SQLite3 database named `jobs.db` and contained in the session directory.
- */path/to/a/directory* – the jobs are stored in the given directory, one file per job (this is the default format used by GC3Pie)
- `sqlite:////path/to/a/file.db` – the jobs are stored in the given SQLite3 database file.
- `mysql://user,passwd@server/dbname` – jobs are stored in table `store` of the specified MySQL database. The DB server and connection credentials (username, password) are also part of the *URL*.

If this option is omitted, GC3Pie's *SessionBasedScript* defaults to storing jobs in the subdirectory `jobs` of the session directory; each job is saved in a separate file.

Exit code A GC3Apps script exits when all jobs are finished, when some error occurred that prevented the script from completing, or when a user interrupts it with `Ctrl+C`

In any case, the exit code of GC3Apps scripts tracks job status (in the following sense). The exitcode is a bitfield; the 4 least-significant bits are assigned a meaning according to the following table:

Bit	Meaning
0	Set if a fatal error occurred: the script could not complete
1	Set if there are jobs in `FAILED` state
2	Set if there are jobs in `RUNNING` or `SUBMITTED` state
3	Set if there are jobs in `NEW` state

This boils down to the following rules:

- exitcode is 0: all jobs are *DONE*, no further action will be taken by the script (which exists immediately if called again on the same session).

- exitcode is 1: an error interrupted the script execution.
- exitcode is 2: all jobs finished, but some are in *FAILED* state.
- exitcode > 3: run the script again to make jobs progress.

The ggamess script

GC3Apps provide a script drive execution of multiple `gamess` jobs each of them with a different input file. It uses the generic *gc3libs.cmdline.SessionBasedScript* framework.

The purpose of GAMESS is to execute *several concurrent runs* of GAMESS each with separate input file. These runs are performed in parallel using every available GC3Pie parameters.

How to run GAMESS on the Grid SSH to *ocikbgtw*, then run the command (it's one single command line, even if it appears broken in several ones in the mail):

```
ggamess.py -A ~/beckya-dmulti.changes.tar.gz -R 2011R3-beckya-dmulti -s "a_session_name" "input_fi
```

The parts in double quotes should be replaced with actual content:

`a_session_name`:

Used for grouping. This is a word of your choosing (e.g., "*test1*", "*control_group*"), used as a label to tag a group of analyses. Multiple concurrent sessions can exist, and they won't interfere one with the other. Again, note that a single session can run many different *.inp* files.

`input_files_or_directories`:

This part consists in the path name of *.inp* files or a directory containing *.inp* files. When a directory is specified, all the *.inp* files contained in it are submitted as GAMESS jobs.

After running, the program will print a short summary of the session (how many jobs running, how many queued, how many finished). Each finished job creates one directory (whose name is equal to the name of the input file, minus the trailing *.inp*), which contains the *.out* and *.dat* files.

For shorter typing, I have defined an alias *ggms* to expand to the above string `ggamess.py -A ... 2011R3-beckya-dmulti`, so you could shorten the command to just:

```
ggms -s "a_session_name" "input_files_or_directories"
```

For instance, to use `ggames.py` to analyse a single *.inp* file you must run:

```
ggms -s "single" dmulti/inp/neutral/dmulti_cc4l.inp
```

while to use `ggamess.py` to run several GAMESS jobs in parallel:

```
ggms -s "multiple" dmulti/inp/neutral
```

Tweaking execution Command-line options (those that start with a dash character '-') can be used to alter the behavior of the `ggamess.py` command:

`-A` *filename.changes.tar.gz*

This selects the file containing your customized version of GAMESS in a format suitable for running in a virtual machine on the Grid. This file should be created following the procedure detailed below.

`-R` *version*

Select a specific version of GAMESS. This should have been installed in the virtual machine within a directory named `gamess-version`; for example, your modified

GAMESS is saved in directory `gamess-2011R3-beckya-dmulti` so the "*version*" string is `2011R3-beckya-dmulti`.

If you omit the `-R` "*version*" part, you get the default GAMESS which is presently 2011R1.

`-s` *session*

Group jobs in a named session; see above.

`-w` *NUM*

Request a running time of at *NUM* hours. If you omit this part, the default is 8 hours.

`-m` *NUM*

Request *NUM* Gigabytes of memory for running each job. GAMESS' memory is measured in words, and each word is 8 bytes; add 1 GB to the total to be safe :-)

Updating the GAMESS code For this you will need to launch the AppPot virtual machine, which is done by running the following command at the command prompt on *ocikbgtw*:

```
apppot-start.sh
```

After a few seconds, you should find yourself at the same `user@rootstrap` prompt that you get on your VirtualBox instance, so you can use the same commands etc.

The only difference of note is that you can exchange files between the AppPot virtual machine and *ocikbgtw* via the *job* directory (whereas it's `/scratch` in VirtualBox). So: files you copy into *job* in the AppPot VM will appear into your home directory on *ocikbgtw*, and conversely files from your home directory on *ocikbgtw* can be read/written as if they were into directory *job* in the AppPot VM.

Once you have compiled a new version of GAMESS that you wish to test, you need to run this command (at the `user@rootstrap` command prompt in the AppPot VM):

```
sudo apppot-snap changes ~/job/beckya-dmulti.changes.tar.gz
```

This will overwrite the file `beckya-dmulti.changes.tar.gz` with the new GAMESS version. If you don't want to overwrite it and instead create another one, just change the filename above (but it *has to* end with the string `.changes.tar.gz`), and the use the new name for the `-R` option to ggamess.py

Exit the AppPot VM by typing `exit` at the command prompt.

The `ggeotop` script

GC3Apps provide a script drive execution of multiple `GEOtop` jobs. It uses the generic *gc3libs.cmdline.SessionBasedScript* framework.

From GEOtop's "read me" file:

```
#
# RUNNING
# Run this simulation by calling the executable (GEOtop_1.223_static)
# and giving the simulation directory as an argument.
#
# EXAMPLE
# ls2:/group/geotop/sim/tmp/000001>./GEOtop_1.223_static ./
#
# TERMINATION OF SIMULATION BY GEOTOP
# When GEOtop terminates due to an internal error, it mostly reports this
# by writing a corresponding file (_FAILED_RUN or _FAILED_RUN.old) in the
# simulation directory. When is terminates sucessfully this file is
# named (_SUCCESSFUL_RUN or _SUCCESSFUL_RUN.old).
#
# RESTARTING SIMULATIONS THAT WERE TERMINATED BY THE SERVER
# When a simulation is started again with the same arguments as described
```

```
# above (RUNNING), then it continues from the last saving point. If
# GEOtop finds a file indicating a successful/failed run, it terminates.
```

Introduction **ggeotop** driver script acan the specified INPUT directories recursively for simulation directories and submit a job for each one found; job progress is monitored and, when a job is done, its output files are retrieved back into the simulation directory itself.

A simulation directory is defined as a directory containing a `geotop.inpts` file, an `in` and an `out` folders.

The `ggeotop` command keeps a record of jobs (submitted, executed and pending) in a session file (set name with the `-s` option); at each invocation of the command, the status of all recorded jobs is updated, output from finished jobs is collected, and a summary table of all known jobs is printed. New jobs are added to the session if new input files are added to the command line.

Options can specify a maximum number of jobs that should be in 'SUBMITTED' or 'RUNNING' state; `ggeotop` will delay submission of newly-created jobs so that this limit is never exceeded.

Options can specify a maximum number of jobs that should be in 'SUBMITTED' or 'RUNNING' state; **ggeotop** will delay submission of newly-created jobs so that this limit is never exceeded.

In more detail, **ggeotop** does the following:

1. Reads the *session* (specified on the command line with the `--session` option) and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.
2. Recursively scans trough `input` folder searching for any valid folder.

   ggeotop will generate a collection of jobs one for each valid input folder. Each job will transfer the input folder to the remote execution node and run `GEOTop`. `GEOTop` reads geotop.inpts files for getting instructions on how to find the input data, what and how to process and where to place generated output results. Extracted from a generic geotop.inpts file:

```
DemFile = "in/dem"
MeteoFile = "in/meteo"
SkyViewFactorMapFile = "in/svf"
SlopeMapFile = "in/slp"
AspectMapFile = "in/asp"

!===========================================
!   DIST OUTPUT
!===========================================
SoilAveragedTempTensorFile = "out/maps/T"
NetShortwaveRadiationMapFile="out/maps/SWnet"
InShortwaveRadiationMapFile="out/maps/SWin"
InLongwaveRadiationMapFile="out/maps/LWin"
SWEMapFile= "out/maps/SWE"
AirTempMapFile = "out/maps/Ta"
```

3. Updates the state of all existing jobs, collects output from finished jobs, and submits new jobs generated in step 2.
4. For each of the terminated jobs, a post-process routine is executed to check and validate the consistency of the generated output. If no `_SUCCESSFUL_RUN` or `_FAILED_RUN` file is found, the related job will be resubmitted together with the current input and output folders. GEOTop is capable of restarting an interrupted claculation by inspecting the intermediate results generated in `out` folder.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option in the *Introduction to session-based scripts* section.)

4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   The program **ggeotop** exits when all jobs have run to completion, i.e., when all valid input folders have been computed.

Execution can be interrupted at any time by pressing `Ctrl+C`. If the execution has been interrupted, it can be resumed at a later stage by calling **ggeotop** with exactly the same command-line options.

Command-line invocation of ggeotop The **ggeotop** script is based on GC3Pie's *session-based script* model; please read also the *Introduction to session-based scripts* section for an introduction to sessions and generic command-line options.

A **ggeotop** command-line is constructed as follows:

1. Each argument (at least one should be specified) is considered as a folder reference.
2. `-x` option is used to specify the path to the GEOtop executable file.

Example 1. The following command-line invocation uses **ggeotop** to run `GEOTop` on all valid input folder found in the recursive check of `input_folder`:

```
$ ggeotop -x /apps/geotop/bin/geotop_1_224_20120227_static ./input_folder
```

Example 2.

```
$ ggeotop --session SAMPLE_SESSION -w 24 -x /apps/geotop/bin/geotop_1_224_20120227_static ./input_
```

In this example, job information is stored into session `SAMPLE_SESSION` (see the documentation of the `--session` option in *Introduction to session-based scripts*). The command above creates the jobs, submits them, and finally prints the following status report:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 10:53:46, 02/28/12)
NEW   0/50    (0.0%)
RUNNING   0/50    (0.0%)
STOPPED   0/50    (0.0%)
SUBMITTED   50/50   (100.0%)
TERMINATED   0/50    (0.0%)
TERMINATING   0/50    (0.0%)
total   50/50   (100.0%)
```

Calling **ggeotop** over and over again will result in the same jobs being monitored;

The `-C` option tells **ggeotop** to continue running until all jobs have finished running and the output files have been correctly retrieved. On successful completion, the command given in *example 2.* above, would print:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 11:05:50, 02/28/12)
NEW   0/50    (0.0%)
RUNNING   0/50    (0.0%)
STOPPED   0/540    (0.0%)
SUBMITTED   0/50    (0.0%)
TERMINATED   50/50   (100.0%)
TERMINATING   0/50    (0.0%)
ok   50/50   (100.0%)
total   50/50   (100.0%)
```

Each job will be named after the folder name (e.g. 000002) (you could see this by passing the `-l` option to **ggeotop**).; each of these jobs will fill the related input folder with the produced outputs.

For each job, the set of output files is automatically retrieved and placed in the locations described below.

Output files for ggeotop Upon successful completion, the output directory of each **ggeotop** job contains:

- the `out` folder will contains what has been produced during the computation of the related job.

Example usage This section contains commented example sessions with **ggeotop**.

Manage a set of jobs from start to end *In typical operation,* one calls **ggeotop** with the `-C` option and lets it manage a set of jobs until completion.

So, to analyse all valid folders under `input_folder`, submitting 200 jobs simultaneously each of them requesting 2GB of memory and 8 hours of *wall-clock time*, one can use the following command-line invocation:

```
$ ggeotop -s example -C 120 -x
/apps/geotop/bin/geotop_1_224_20120227_static -w 8 input_folder
```

The `-s example` option tells **ggeotop** to store information about the computational jobs in the `example.jobs` directory.

The `-C 120` option tells **ggeotop** to update job state every 120 seconds; output from finished jobs is retrieved and new jobs are submitted at the same interval.

The above command will start by printing a status report like the following:

```
Status of jobs in the 'example.csv' session:
SUBMITTED   1/1 (100.0%)
```

It will continue printing an updated status report every 120 seconds until the requested parameter range has been computed.

In GC3Pie terminology when a job is finished and its output has been successfully retrieved, the job is marked as `TERMINATED`:

```
Status of jobs in the 'example.csv' session:
TERMINATED   1/1 (100.0%)
```

Using GC3Pie utilities GC3Pie comes with a set of generic utilities that could be used as a complemet to the **ggeotop** command to better manage a entire session execution.

gkill: cancel a running job To cancel a running job, you can use the command **gkill**. For instance, to cancel *job.16*, you would type the following command into the terminal:

```
gkill job.16
```

or:

```
gkill -s example job.16
```

gkill could also be used to cancel jobs in a given state

```
gkill -s example -l UNKNOWN
```

> **Warning:** *There's no way to undo a cancel operation!* Once you have issued a **gkill** command, the job is deleted and it cannot be resumed. (You can still re-submit it with **gresub**, though.)

ginfo: accessing low-level details of a job It is sometimes necessary, for debugging purposes, to print out all the details about a job; the **ginfo** command does just that: prints all the details that GC3Utils know about a single job.

For instance, to print out detailed information about *job.13* in session *example*, you would type

```
ginfo -s example job.13
```

For a job in `RUNNING` or `SUBMITTED` state, only little information is known: basically, where the job is running, and when it was started:

```
$ ginfo -s example job.13
job.13
    cores: 2
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:05 2012
        Submitted to 'wsl' at Tue May 15 09:52:05 2012
        RUNNING at Tue May 15 10:07:39 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/116613370683251353308673
    lrms_jobname: GC3Pie_00002
    original_exitcode: -1
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069259.18
    stderr_filename: ggeotop.log
    stdout_filename: ggeotop.log
    timestamp:
        RUNNING: 1337069259.18
        SUBMITTED: 1337068325.26
    unknown_iteration: 0
    used_cputime: 1380
    used_memory: 3382706
```

If you omit the job number, information about *all* jobs in the session will be printed.

Most of the output is only useful if you are familiar with GC3Utils inner working. Nonetheless, **ginfo** output is definitely something you should include in any report about a misbehaving job!

For a finished job, the information is more complete and can include error messages in case the job has failed:

```
$ ginfo -c -s example job.13
job.13
    _arc0_state_last_checked: 1337069259.18
    _exitcode: 0
    _signal: None
    _state: TERMINATED
    cores: 2
    download_dir: /data/geotop/results/00002
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:04 2012
        Submitted to 'wsl' at Tue May 15 09:52:04 2012
        TERMINATING at Tue May 15 10:07:39 2012
        Final output downloaded to '/data/geotop/results/00002'
        TERMINATED at Tue May 15 10:07:43 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/11441337068324584585032
    lrms_jobname: GC3Pie_00002
    original_exitcode: 0
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069263.13
    stderr_filename: ggeotop.log
    stdout_filename: ggeotop.log
    timestamp:
        SUBMITTED: 1337068324.87
        TERMINATED: 1337069263.13
        TERMINATING: 1337069259.18
    unknown_iteration: 0
    used_cputime: 360
    used_memory: 3366977
    used_walltime: 300
```

With option `-v`, **ginfo** output is even more verbose and complete, and includes information about the application itself, the input and output files, plus some backend-specific information

```
$ ginfo -c -s example job.13
job.13
  arguments: 00002
  changed: False
  environment:
  executable: geotop_static
  executables: geotop_static
  execution:
      _arc0_state_last_checked: 1337069259.18
      _exitcode: 0
      _signal: None
      _state: TERMINATED
      cores: 2
      download_dir: /data/geotop/results/00002
      execution_targets: hera.wsl.ch
      log:
          SUBMITTED at Tue May 15 09:52:04 2012
          Submitted to 'wsl' at Tue May 15 09:52:04 2012
          TERMINATING at Tue May 15 10:07:39 2012
          Final output downloaded to '/data/geotop/results/00002'
          TERMINATED at Tue May 15 10:07:43 2012
      lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/11441337068324584585032
      lrms_jobname: GC3Pie_00002
      original_exitcode: 0
      queue: smscg.q
      resource_name: wsl
      state_last_changed: 1337069263.13
      stderr_filename: ggeotop.log
      stdout_filename: ggeotop.log
      timestamp:
          SUBMITTED: 1337068324.87
          TERMINATED: 1337069263.13
          TERMINATING: 1337069259.18
      unknown_iteration: 0
      used_cputime: 360
      used_memory: 3366977
      used_walltime: 300
  jobname: GC3Pie_00002
  join: True
  output_base_url: None
  output_dir: /data/geotop/results/00002
  outputs:
      @output.list: file, , @output.list, None, None, None, None
      ggeotop.log: file, , ggeotop.log, None, None, None, None
  persistent_id: job.1698503
  requested_architecture: x86_64
  requested_cores: 2
  requested_memory: 4
  requested_walltime: 4
  stderr: None
  stdin: None
  stdout: ggeotop.log
  tags: APPS/EARTH/GEOTOP
```

The `grosetta` and `gdocking` scripts

GC3Apps provide two scripts to drive execution of applications (*protocols*, in Rosetta terminology) from the Rosetta bioinformatics suite.

The purpose of **`grosetta`** and **`gdocking`** is to execute *several concurrent runs* of minirosetta or docking_protocol on a set of input files, and collect the generated output. These runs are performed in parallel using

every available GC3Pie *resource*; you can of course control how many runs should be executed and select what output files you want from each one.

The script **grosetta** is a relatively generic front-end that executes the minirosetta program by default (but a different application can be chosen with the `-x` *command-line option*). The **gdocking** script is specialized for running Rosetta's docking_protocol program.

Introduction The **grosetta** and **gdocking** execute *several runs* of minirosetta or docking_protocol on a set of input files, and collect the generated output. These runs are performed in parallel, up to a limit that can be configured with the `-J` *command-line option*. You can of course control how many runs should be executed and select what output files you want from each one.

Note: The **grosetta** and **gdocking** scripts are very similar in usage. In the following, whatever is written about **grosetta** applies to **gdocking** as well; the differences will be pointed out on a case-by-case basis.

In more detail, **grosetta** does the following:

1. Reads the *session* (specified on the command line with the `--session` option) and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.
2. Scans the input file names given on the command-line, and generates a number of identical computational jobs, all running the same Rosetta program on the same set of input files. The objective is to compute a specified number *P* of decoys of any given PDB file.

   The number *P* of wanted decoys can be set with the `--total-decoys` option (see below). The option `--decoys-per-job` can set the number of decoys that each computational job can compute; this should be a guessed based on the maximum allowed run time of each job and the time taken by the Rosetta protocol to compute a single decoy.
3. Updates the state of all existing jobs, collects output from finished jobs, and submits new jobs generated in step 2.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option in the *Introduction to session-based scripts* section.)
4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   The program **grosetta** exits when all jobs have run to completion, i.e., when the wanted number of decoys have been computed.

   Execution can be interrupted at any time by pressing `Ctrl+C`. If the execution has been interrupted, it can be resumed at a later stage by calling **grosetta** with exactly the same command-line options.

The **gdocking** program works in exactly the same way, with the important exception that **gdocking** uses a separate Rosetta docking_protocol program invocation *per input file*.

Command-line invocation of grosetta The **grosetta** script is based on GC3Pie's *session-based script* model; please read also the *Introduction to session-based scripts* section for an introduction to sessions and generic command-line options.

A **grosetta** command-line is constructed as follows:

1. The 1st argument is the *flags* file, containing options to pass to every executed Rosetta program;
2. then follows any number of input files (copied from your PC to the execution site);
3. then a literal colon character `:`;
4. finally, you can list any number of output file patterns (copied back from the execution site to your PC); wildcards (e.g., `*.pdb`) are allowed, but you must enclose them in quotes. Note that:
   - you can omit the output files: the default is `"*.pdb" "*.sc" "*.fasc"`
   - if you omit the output files patterns, omit the colon as well

Example 1. The following command-line invocation uses **grosetta** to run minirosetta on the molecule files `1bjpA.pdb`, `1ca7A.pdb`, and `1cgqA.pdb`. The `flags` file (1st command-line argument) is a text file containing options to pass to the actual minirosetta program. Additional input files are specified on the command line between the `flags` file and the PDB input files.

```
$ grosetta flags alignment.filt query.fasta query.psipred_ss2 boinc_aaquery03_05.200_v1_3.

You can see that the listing of output patterns has been omitted,
so `grosetta`:command: will use the default and retrieve all
`*.pdb`:file:, `*.sc`:file: and `*.fasc`:file: files.
```

There will be a number of *identical* jobs being executed as a result of a **grosetta** or **gdocking** invocation; this number depends on the ratio of the values given to options `-P` and `-p`:

-P NUM, --total-decoys NUM Compute *NUM* decoys per input file.

-p NUM, --decoys-per-job NUM Compute *NUM* decoys in a single job (default: 1). This parameter should be tuned so that the running time of a single job does not exceed the maximum wall-clock time (see the `--wall-clock-time` command-line option in *Introduction to session-based scripts*).

If you omit `-P` and `-p`, they both default to 1, i.e., one job will be created (as in the *example 1.* above).

Example 2. The following command-line invocation will run 3 parallel instances of minirosetta, each of which generates 2 decoys (save the last one, which only generates 1 decoy) of the molecule described in file `1bjpA.pdb`:

```
$ grosetta --session SAMPLE_SESSION --total-decoys 5 --decoys-per-job 2 flags alignment.filt
```

In this example, job information is stored into session `SAMPLE_SESSION` (see the documentation of the `--session` option in *Introduction to session-based scripts*). The command above creates the jobs, submits them, and finally prints the following status report:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 10:53:46, 02/28/12)
        NEW   0/3    (0.0%)
    RUNNING   0/3    (0.0%)
    STOPPED   0/3    (0.0%)
  SUBMITTED   3/3   (100.0%)
 TERMINATED   0/3    (0.0%)
TERMINATING   0/3    (0.0%)
      total   3/3   (100.0%)
```

Note that the status report counts the number of *jobs in the session*, not the total number of decoys being generated. (Feel free to report this as a bug.)

Calling `grosetta` over and over again will result in the same jobs being monitored; to create new jobs, change the command line and raise the value for `-P` or `-p`. (To completely erase an existing session and start over, use the `--new-session` option, as per *session-based script* documentation.)

The `-C` option tells **grosetta** to continue running until all jobs have finished running and the output files have been correctly retrieved. On successful completion, the command given in *example 2.* above, would print:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 11:05:50, 02/28/12)
        NEW   0/3    (0.0%)
    RUNNING   0/3    (0.0%)
    STOPPED   0/3    (0.0%)
  SUBMITTED   0/3    (0.0%)
 TERMINATED   3/3   (100.0%)
TERMINATING   0/3    (0.0%)
         ok   3/3   (100.0%)
      total   3/3   (100.0%)
```

The three jobs are named `0--1`, `2--3` and `4--5` (you could see this by passing the `-l` option to **grosetta**); each of these jobs will create an output directory named after the job.

In general, **grosetta** jobs are named *N--M* with *N* and *M* being two integers from 0 up to the value specified with option `--total-decoys`. Jobs generated by **gdocking** are instead named after the input file, with a *.N--M* suffix added.

For each job, the set of output files is automatically retrieved and placed in the locations described below.

Note: The naming and contents of output files differ between **grosetta** and **gdocking**. Refer to the appropriate section below!

Output files for grosetta Upon successful completion, the output directory of each **grosetta** job contains:

- A copy of the *input* PDB files;
- Additional `.pdb` files named `S_`*random string*`.pdb`, generated by minirosetta during its run;
- A file `score.sc`;
- Files `minirosetta.static.log`, `minirosetta.static.stdout.txt` and `minirosetta.static.stderr.txt`.

The `minirosetta.static.log` file contains the output log of the minirosetta execution. For each of the `S_*.pdb` files above, a line like the following should be present in the log file (the file name and number of elapsed seconds will of course vary!):

```
protocols.jd2.JobDistributor: S_1CA7A_1_0001 reported success in 124 seconds
```

The `minirosetta.static.stdout.txt` contains a copy of the minirosetta output log, plus the output of the wrapper script. In case of successful minirosetta run, the last line of this file will read:

```
minirosetta.static: All done, exitcode: 0
```

Output files for gdocking Execution of `gdocking` yields the following output:

- For each `.pdb` input file, a `.decoys.tar` file (e.g., for `1bjpa.pdb` input, a `1bjpa.decoys.tar` output is produced), which contains the `.pdb` files of the decoys produced by **gdocking**.
- For each successful job, a *.N–M* directory: e.g., for the `1bjpa.1--2` job, a `1bjpa.1--2/` directory is created, with the following content:
  - `docking_protocol.log`: output of Rosetta's `docking_protocol` program;
  - `docking_protocol.stderr.txt`, `docking_protocol.stdout.txt`: obvoius meaning. The "stdout" file contains a copy of the `docking_protocol.log` contents, plus the output from the wrapper script.
  - `docking_protocol.tar.gz`: the `.pdb` decoy files produced by the job.

The following scheme summarizes the location of **gdocking** output files:

```
(directory where gdocking is run)/
  |
  +- file1.pdb      Original input file
  |
  +- file1.N--M/    Directory collecting job outputs from job file1.N--M
  |    |
  |    +- docking_protocol.tar.gz
  |    +- docking_protocol.log
  |    +- docking_protocol.stderr.txt
  |    ... etc
  |
  +- file1.N--M.fasc   FASC file for decoys N to M [1]
  |
  +- file1.decoys.tar  tar archive of PDB file of all decoys
  |                    generated corresponding to 'file1.pdb' [2]
```

```
  |
  ...
```

Let *P* be the total number of decoys (the argument to the `-P` option), and *p* be the number of decoys per job (argument to the `-p` option). Then you would get in a single directory:

1. *(P/p)* different `.fasc` files, corresponding to the *(P/p)* jobs;
2. *P* different `.pdb` files, named `a_file.0.pdb` to `a_file.` *(P-1)* `.pdb`

Example usage This section contains commented example sessions with **grosetta**. All the files used in this example are available in the GC3Pie Rosetta test directory (courtesy of Lars Malmstroem).

Manage a set of jobs from start to end *In typical operation,* one calls **grosetta** with the `-C` option and lets it manage a set of jobs until completion.

So, to generate one decoy from a set of given input files, one can use the following command-line invocation:

```
$ grosetta -s example -C 120 -P 1 -p 1 \
    flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.pdb \
    2fltA.pdb 2fm7A.pdb 2op8A.pdb 2ormA.pdb 2os5A.pdb \
    3c6vA.pdb
```

The `-s example` option tells **grosetta** to store information about the computational jobs in the `example.jobs` directory.

The `-C 120` option tells **grosetta** to update job state every 120 seconds; output from finished jobs is retrieved and new jobs are submitted at the same interval.

The `-P 1` and `-p 1` options set the total number of decoys to compute and the maximum number of decoys that a single computational job can handle. These values can be arbitrarily high (however the *p* value should be such that the computational job can actually compute that many decoys in the allotted *wall-clock time*).

The above command will start by printing a status report like the following:

```
Status of jobs in the 'example.csv' session:
 SUBMITTED   1/1 (100.0%)
```

It will continue printing an updated status report every 120 seconds until the requested number of decoys (set by the `-P` option) has been computed.

In GC3Pie terminology when a job is finished and its output has been successfully retrieved, the job is marked as `TERMINATED`:

```
Status of jobs in the 'example.csv' session:
 TERMINATED   1/1 (100.0%)
```

Managing a session by repeated `grosetta` invocation We now show how one can obtain the same result by calling **grosetta** multiple times (there could be hours of interruption between one invocation and the next one).

Note: This is not the typical mode of operating with **grosetta**, but may still be useful in certain settings.

---

1. Create a session (1 job only, since no `-P` option is given); the session name is chosen with the `-s` (short for `--session`) option. You should take care of re-using the same session name with subsequent commands.

```
    $ grosetta -s example flags alignment.filt query.fasta \
        query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
        boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.pdb \
        2fltA.pdb 2fm7A.pdb 2op8A.pdb 2ormA.pdb 2os5A.pdb
```

```
Status of jobs in the 'example.csv' session:
 SUBMITTED   1/1 (100.0%)
```

2. Now we call **grosetta** again, and request that 3 decoys be computed starting from a single PDB file (`--total-decoys 3` on the command line). Since we are submitting a single PDB file, the 3 decoys will be computed all in a single run, so the `--decoys-per-job` option will have value `3`.

```
$ grosetta -s example --total-decoys 3 --decoys-per-job 3 \
    flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 3c6vA.pdb
Status of jobs in the 'example.csv' session:
 SUBMITTED   3/3 (100.0%)
```

   Note that 3 jobs were submitted: **grosetta** interprets the `--total-decoys` option globally, and adds one job to compute the 2 missing decoys from the file set from step 1. (This is currently a limitation of **grosetta**)

   From here on, one could simply run `grosetta -C 120` and let it manage the session until completion of all jobs, as in the example *Manage a set of jobs from start to end* above. For the sake of showing how the use of several command-line options of **grosetta**, we shall further show how manage the session by repeated separate invocations.

3. Next step is to monitor the session, so we add the command-line option `-l` which tells **grosetta** to list all the jobs with their status. Also note that we keep the `-s example` option to tell **grosetta** that we would like to operate on the session named *example*.

   All non-option arguments can be omitted: as long as the total number of decoys is unchanged, they're not needed.

```
$ grosetta -s example -l
Decoys Nr.       State (JobID)       Info
================================================================================
0--1             RUNNING (job.766)   Running at Mon Dec 20 19:32:08 2010
2--3             RUNNING (job.767)   Running at Mon Dec 20 19:33:23 2010
0--2             RUNNING (job.768)   Running at Mon Dec 20 19:33:43 2010
```

   Without the `-l` option only a summary of job statuses is presented:

```
$ grosetta -s example
Status of jobs in the 'grosetta.csv' session:
 RUNNING     3/3 (100.0%)
```

   Alternatively, we can keep the command line arguments used in the previous invocation: they will be ignored since they do not add any new job (the number of decoys to compute is always 1):

```
$ grosetta -s example -l flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.pdb \
    2fltA.pdb 2fm7A.pdb 2op8A.pdb 2ormA.pdb 2os5A.pdb \
    3c6vA.pdb
Decoys Nr.       State (JobID)       Info
================================================================================
0--1             RUNNING (job.766)
2--3             RUNNING (job.767)   Running at Mon Dec 20 19:33:23 2010
0--2             RUNNING (job.768)   Running at Mon Dec 20 19:33:43 2010
```

   Note that the `-l` option is available also in combination with the `-C` option (see *Manage a set of jobs from start to end*).

4. Calling `grosetta` again when jobs are done triggers automated download of the results:

```
$ ../grosetta.py
File downloaded:
gsiftp://idgc3grid01.uzh.ch:2811/jobs/214661292869757468202765/minirosetta.static.stdout.txt
```

```
File downloaded:
gsiftp://idgc3grid01.uzh.ch:2811/jobs/214661292869757468202765/minirosetta.static.log
 ...
File downloaded:
gsiftp://idgc3grid01.uzh.ch:2811/jobs/214661292869757468202765/.arc/input
Status of jobs in the 'grosetta.csv' session:
 TERMINATED  1/1 (100.0%)
 ok          1/1 (100.0%)
```

The `-l` option comes handy to see what directory contains the job output:

```
$ grosetta -l
Decoys Nr.       State (JobID)         Info
================================================================================
0--1             TERMINATED (job.766)  Output retrieved into directory '/tmp/0--1'
```

The `gcrypto` script

GC3Apps provide a script drive execution of multiple `gnfs-cmd` jobs each of them with a different parameter set. Allotogehter they form a single crypto simulation of a large parameter space. It uses the generic *gc3libs.cmdline.SessionBasedScript* framework.

The purpose of **`gcrypto`** is to execute *several concurrent runs* of `gnfs-cmd` on a parameter set. These runs are performed in parallel using every available GC3Pie *resource*; you can of course control how many runs should be executed and select what output files you want from each one.

Introduction Like in a *for*-loop, the **`gcrypto`** driver script takes as input three mandatory arguments:

1. RANGE_START: initial value of the range (e.g., 800000000)
2. RANGE_END: final value of the range (e.g., 1200000000)
3. SLICE: extent of the range that will be examined by a single job (e.g., 1000)

For example:

```
# gcrypto 800000000 1200000000 1000
```

will produce 400000 jobs; the first job will perform calculations on the range 800000000 to 800000000+1000, the 2nd one will do the range 800001000 to 800002000, and so on.

Inputfile archive location (e.g. *lfc://lfc.smscg.ch/crypto/lacal/input.tgz*) can be specified with the '-i' option. Otherwise a default filename 'input.tgz' will be searched in current directory.

Job progress is monitored and, when a job is done, output is retrieved back to submitting host in folders named: `RANGE_START + (SLICE * ACTUAL_STEP)` Where `ACTUAL_STEP` correspond to the position of the job in the overall execution.

The **`gcrypto`** command keeps a record of jobs (submitted, executed and pending) in a session file (set name with the '-s' option); at each invocation of the command, the status of all recorded jobs is updated, output from finished jobs is collected, and a summary table of all known jobs is printed. New jobs are added to the session if new input files are added to the command line.

Options can specify a maximum number of jobs that should be in 'SUBMITTED' or 'RUNNING' state; **`gcrypto`** will delay submission of newly-created jobs so that this limit is never exceeded.

The **`gcrypto`** execute *several runs* of `gnfs-cmd` on a parameter set, and collect the generated output. These runs are performed in parallel, up to a limit that can be configured with the `-J` *command-line option*. You can of course control how many runs should be executed and select what output files you want from each one.

In more detail, **`gcrypto`** does the following:

1. Reads the *session* (specified on the command line with the `--session` option) and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.

2. Divide the initial parameter range, given in the command-line, into chunks taking the `-J` value as a reference. So from a coomand line argument like the following:

```
$ gcrypto  800000000 1200000000 1000 -J 200
```

   gcrypto will generate an initial chunks of 200 jobs starting from the initial range 800000000 incrementing of 1000. All jobs will run `gnfs-cmd` on a specific parameter set (e.g. 800000000, 800001000, 800002000, ...). **gcrypto** will keep constant the number of simulatenous jobs running retrieving those terminated and submitting new ones untill the whole parameter range has been computed.

3. Updates the state of all existing jobs, collects output from finished jobs, and submits new jobs generated in step 2.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option in the *Introduction to session-based scripts* section.)

4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   The program **gcrypto** exits when all jobs have run to completion, i.e., when the whole paramenter range has been computed.

Execution can be interrupted at any time by pressing `Ctrl+C`. If the execution has been interrupted, it can be resumed at a later stage by calling **gcrypto** with exactly the same command-line options.

gcrypto requires a number of default input files common to every submited job. This list of input files is automatically fetched by **gcrypto** from a default storage repository. Those files are:

```
gnfs-lasieve6
M1019
M1019.st
M1037
M1037.st
M1051
M1051.st
M1067
M1067.st
M1069
M1069.st
M1093
M1093.st
M1109
M1109.st
M1117
M1117.st
M1123
M1123.st
M1147
M1147.st
M1171
M1171.st
M8e_1200
M8e_1500
M8e_200
M8e_2000
M8e_2500
M8e_300
M8e_3000
M8e_400
M8e_4200
M8e_600
M8e_800
tdsievemt
```

When **gcrypto** has to be executed with a different set of input files, an additional command line argument `--input-files` could be used to specify the locatin of a `tar.gz` archive containing the input files that `gnfs-cmd` will expect. Similarly, when a different version of gnfs-cmd command needs to be used, the command line argument `--gnfs-cmd` could be used to specify the location of the `gnfs-cmd` to be used.

Command-line invocation of gcrypto The **gcrypto** script is based on GC3Pie's *session-based script* model; please read also the *Introduction to session-based scripts* section for an introduction to sessions and generic command-line options.

A **gcrypto** command-line is constructed as follows: Like a *for*-loop, the **gcrypto** driver script takes as input three mandatory arguments:

1. *RANGE_START*: initial value of the range (e.g., 800000000)
2. *RANGE_END*: final value of the range (e.g., 1200000000)
3. *SLICE*: extent of the range that will be examined by a single job (e.g., 1000)

Example 1. The following command-line invocation uses **gcrypto** to run `gnfs-cmd` on the parameter set ranging from 800000000 to 1200000000 with an increment of 1000.

```
$ gcrypto 800000000 1200000000 1000
```

In this case **gcrypto** will use the default values for determine the chunks size from the default value of the `-J` option (default value is 50 simulatenous jobs).

Example 2.

```
$ gcrypto --session SAMPLE_SESSION -c 4 -w 4 -m 8 800000000 1200000000 1000
```

In this example, job information is stored into session `SAMPLE_SESSION` (see the documentation of the `--session` option in *Introduction to session-based scripts*). The command above creates the jobs, submits them, and finally prints the following status report:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 10:53:46, 02/28/12)
NEW   0/50    (0.0%)
RUNNING   0/50    (0.0%)
STOPPED   0/50    (0.0%)
SUBMITTED   50/50   (100.0%)
TERMINATED   0/50    (0.0%)
TERMINATING   0/50    (0.0%)
total   50/50   (100.0%)
```

Note that the status report counts the number of *jobs in the session*, not the total number of jobs that would correspond to the whole parameter range. (Feel free to report this as a bug.)

Calling **gcrypto** over and over again will result in the same jobs being monitored;

The `-C` option tells **gcrypto** to continue running until all jobs have finished running and the output files have been correctly retrieved. On successful completion, the command given in *example 2.* above, would print:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 11:05:50, 02/28/12)
NEW   0/400k    (0.0%)
RUNNING   0/400k    (0.0%)
STOPPED   0/400k    (0.0%)
SUBMITTED   0/400k    (0.0%)
TERMINATED   50/400k   (100.0%)
TERMINATING   0/400k    (0.0%)
ok   400k/400k   (100.0%)
total   400k/400k   (100.0%)
```

Each job will be named after the parameter range it has computed (e.g. 800001000, 800002000, ...) (you could see this by passing the `-l` option to **gcrypto**); each of these jobs will create an output directory named after the job.

For each job, the set of output files is automatically retrieved and placed in the locations described below.

Output files for `gcrypto` Upon successful completion, the output directory of each **`gcrypto`** job contains:

- a number of `.tgz` files each of them correspondin to a step within the execution of the `gnfs-cmd` command.
- A log file named `gcrypto.log` containing both the *stdout* and the *stderr* of the `gnfs-cmd` execution.

Note: The number of `.tgz` files may depend on whether the execution of the `gnfs-cmd` command has completed or not (e.g. jobs may be killed by the batch system when exausting requested resources)

Example usage This section contains commented example sessions with **`gcrypto`**.

Manage a set of jobs from start to end *In typical operation,* one calls **`gcrypto`** with the `-C` option and lets it manage a set of jobs until completion.

So, to compute a whole parameter range from 800000000 to 1200000000 with an increment of 1000, submitting 200 jobs simultaneously each of them requesting 4 computing cores, 8GB of memory and 4 hours of *wall-clock time*, one can use the following command-line invocation:

```
$ gcrypto -s example -C 120 -J 200 -c 4 -w 4 -m 8 800000000 1200000000 1000
```

The `-s example` option tells **`gcrypto`** to store information about the computational jobs in the `example.jobs` directory.

The `-C 120` option tells **`gcrypto`** to update job state every 120 seconds; output from finished jobs is retrieved and new jobs are submitted at the same interval.

The above command will start by printing a status report like the following:

```
Status of jobs in the 'example.csv' session:
SUBMITTED   1/1 (100.0%)
```

It will continue printing an updated status report every 120 seconds until the requested parameter range has been computed.

In GC3Pie terminology when a job is finished and its output has been successfully retrieved, the job is marked as `TERMINATED`:

```
Status of jobs in the 'example.csv' session:
TERMINATED   1/1 (100.0%)
```

Using GC3Pie utilities GC3Pie comes with a set of generic utilities that could be used as a complemet to the **`gcrypto`** command to better manage a entire session execution.

`gkill`: cancel a running job To cancel a running job, you can use the command **`gkill`**. For instance, to cancel *job.16*, you would type the following command into the terminal:

```
gkill job.16
```

or:

```
gkill -s example job.16
```

gkill could also be used to cancel jobs in a given state

```
gkill -s example -l UNKNOWN
```

Warning: *There's no way to undo a cancel operation!* Once you have issued a **`gkill`** command, the job is deleted and it cannot be resumed. (You can still re-submit it with **`gresub`**, though.)

`ginfo`: accessing low-level details of a job It is sometimes necessary, for debugging purposes, to print out all the details about a job; the **`ginfo`** command does just that: prints all the details that GC3Utils know about a single job.

For instance, to print out detailed information about *job.13* in session *example*, you would type

```
ginfo -s example job.13
```

For a job in `RUNNING` or `SUBMITTED` state, only little information is known: basically, where the job is running, and when it was started:

```
$ ginfo -s example job.13
job.13
    cores: 2
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:05 2012
        Submitted to 'wsl' at Tue May 15 09:52:05 2012
        RUNNING at Tue May 15 10:07:39 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/116613370683251353308673
    lrms_jobname: LACAL_800001000
    original_exitcode: -1
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069259.18
    stderr_filename: gcrypto.log
    stdout_filename: gcrypto.log
    timestamp:
        RUNNING: 1337069259.18
        SUBMITTED: 1337068325.26
    unknown_iteration: 0
    used_cputime: 1380
    used_memory: 3382706
```

If you omit the job number, information about *all* jobs in the session will be printed.

Most of the output is only useful if you are familiar with GC3Utils inner working. Nonetheless, **`ginfo`** output is definitely something you should include in any report about a misbehaving job!

For a finished job, the information is more complete and can include error messages in case the job has failed:

```
$ ginfo -c -s example job.13
job.13
    _arc0_state_last_checked: 1337069259.18
    _exitcode: 0
    _signal: None
    _state: TERMINATED
    cores: 2
    download_dir: /data/crypto/results/example.out/8000001000
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:04 2012
        Submitted to 'wsl' at Tue May 15 09:52:04 2012
        TERMINATING at Tue May 15 10:07:39 2012
        Final output downloaded to '/data/crypto/results/example.out/8000001000'
        TERMINATED at Tue May 15 10:07:43 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/11441337068324584585032
    lrms_jobname: LACAL_800001000
    original_exitcode: 0
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069263.13
    stderr_filename: gcrypto.log
    stdout_filename: gcrypto.log
    timestamp:
```

```
        SUBMITTED: 1337068324.87
        TERMINATED: 1337069263.13
        TERMINATING: 1337069259.18
    unknown_iteration: 0
    used_cputime: 360
    used_memory: 3366977
    used_walltime: 300
```

With option `-v`, **`ginfo`** output is even more verbose and complete, and includes information about the application itself, the input and output files, plus some backend-specific information:

```
$ ginfo -c -s example job.13
job.13
  arguments: 800000800, 100, 2, input.tgz
  changed: False
  environment:
  executable: gnfs-cmd
  executables: gnfs-cmd
  execution:
      _arc0_state_last_checked: 1337069259.18
      _exitcode: 0
      _signal: None
      _state: TERMINATED
      cores: 2
      download_dir: /data/crypto/results/example.out/8000001000
      execution_targets: hera.wsl.ch
      log:
          SUBMITTED at Tue May 15 09:52:04 2012
          Submitted to 'wsl' at Tue May 15 09:52:04 2012
          TERMINATING at Tue May 15 10:07:39 2012
          Final output downloaded to '/data/crypto/results/example.out/8000001000'
          TERMINATED at Tue May 15 10:07:43 2012
      lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/11441337068324584585032
      lrms_jobname: LACAL_800001000
      original_exitcode: 0
      queue: smscg.q
      resource_name: wsl
      state_last_changed: 1337069263.13
      stderr_filename: gcrypto.log
      stdout_filename: gcrypto.log
      timestamp:
          SUBMITTED: 1337068324.87
          TERMINATED: 1337069263.13
          TERMINATING: 1337069259.18
      unknown_iteration: 0
      used_cputime: 360
      used_memory: 3366977
      used_walltime: 300
  inputs:
      srm://dpm.lhep.unibe.ch/dpm/lhep.unibe.ch/home/crypto/gnfs-cmd_20120406: gnfs-cmd
      srm://dpm.lhep.unibe.ch/dpm/lhep.unibe.ch/home/crypto/lacal_input_files.tgz: input.tgz
  jobname: LACAL_800000900
  join: True
  output_base_url: None
  output_dir: /data/crypto/results/example.out/8000001000
  outputs:
      @output.list: file, , @output.list, None, None, None, None
      gcrypto.log: file, , gcrypto.log, None, None, None, None
  persistent_id: job.1698503
  requested_architecture: x86_64
  requested_cores: 2
  requested_memory: 4
  requested_walltime: 4
```

```
  stderr: None
  stdin: None
  stdout: gcrypto.log
  tags: APPS/CRYPTO/LACAL-1.0
```

The GC3Utils software

The GC3Utils are lower-level commands, provided to perform common operations on jobs, regardless of their type or the application they run.

For instance, GC3Utils provide commands to obtain the list and status of computational resources (**gservers**); to clear the list of jobs from old and failed ones (**gclean**); to get detailed information on a submitted job (**ginfo**, mainly for debugging purposes).

This chapter is a tutorial for the GC3Utils command-line utilities.

If you find a technical term whose meaning is not clear to you, please look it up in the *Glossary*. (But feel free to ask on the GC3Pie mailing list if it's still unclear!)

Contents

gsession: manage sessions

All jobs managed by one of the GC3Pie scripts are grouped into sessions; information related of a session is stored into a directory. The **gsession** command allows you to show the jobs related to a specific session, to abort the session or to completely delete it.

The **gsession** accept two mandatory arguments: *command* and *session*. *command* must be one of:

list list jobs related to the session.

log show the session history.

abort kill all jobs related to the session.

delete abort the session and delete the session directory from disk.

For instance, if you want to check the status of the main tasks of a session, just run:

```
$ gsession list SESSION_DIRECTORY
+------------------------------+--------------------------+-------+-----------------------------
| JobID                        | Job name                 | State | Info
+------------------------------+--------------------------+-------+-----------------------------
| ParallelTaskCollection.1140527 | ParallelTaskCollection-N1 | NEW   | NEW at Fri Feb 22 16:39:34
+------------------------------+--------------------------+-------+-----------------------------
```

This command will only show the *top-level tasks*, e.g. the main tasks created by the GC3 script. If you want to see **all** the tasks related to the session run the command with the option `-r`:

```
$ gsession list SESSION_DIRECTORY -r
+---------------------------------------+--------------------------+------------+-------------
| JobID                                 | Job name                 | State      | Info
+---------------------------------------+--------------------------+------------+-------------
| ParallelTaskCollection.1140527        | ParallelTaskCollection-N1 | NEW        | NEW at Fri Fe
|   WarholizeWorkflow.1140528           | WarholizedWorkflow       | RUNNING    | RUNNING at Fr
|     GrayScaleConvertApplication.1140529 |                        | TERMINATED | TERMINATED at
|     TricolorizeMultipleImages.1140530 | Warholizer_Parallel      | NEW        |
|       TricolorizeImage.1140531        | TricolorizeImage         | NEW        |
|         CreateLutApplication.1140532  |                          | NEW        |
|       TricolorizeImage.1140533        | TricolorizeImage         | NEW        |
|         CreateLutApplication.1140534  |                          | NEW        |
|       TricolorizeImage.1140535        | TricolorizeImage         | NEW        |
|         CreateLutApplication.1140536  |                          | NEW        |
|       TricolorizeImage.1140537        | TricolorizeImage         | NEW        |
|         CreateLutApplication.1140538  |                          | NEW        |
+---------------------------------------+--------------------------+------------+-------------
```

To have the full history of the session run *gsession log*:

```
$ gsession log SESSION_DIRECTORY
Feb 22 16:39:01 GrayScaleConvertApplication.1140529: Submitting to 'hobbes' at Fri Feb 22 16:39:01
Feb 22 16:39:08 GrayScaleConvertApplication.1140529: RUNNING
Feb 22 16:39:08 GrayScaleConvertApplication.1140529: SUBMITTED
Feb 22 16:39:08 GrayScaleConvertApplication.1140529: Submitted to 'hobbes' at Fri Feb 22 16:39:08
Feb 22 16:39:08 WarholizeWorkflow.1140528: SUBMITTED
Feb 22 16:39:24 GrayScaleConvertApplication.1140529: TERMINATING
Feb 22 16:39:25 WarholizeWorkflow.1140528: RUNNING
Feb 22 16:39:25 ParallelTaskCollection.1140527: RUNNING
Feb 22 16:39:25 GrayScaleConvertApplication.1140529: Final output downloaded to 'Warholized.lena.
Feb 22 16:39:25 GrayScaleConvertApplication.1140529: TERMINATED
Feb 22 16:39:34 WarholizeWorkflow.1140528: NEW
Feb 22 16:39:34 ParallelTaskCollection.1140527: NEW
Feb 22 16:39:34 WarholizeWorkflow.1140528: RUNNING
```

To abort a session, run the *gsession abort* command:

```
$ gsession abort SESSION_DIRECTORY
```

This will kill all the running jobs and retrieve the results of the terminated jobs, but will leave the session directory untouched. To also delete the session directory, run *gsession delete*:

```
$ gsession delete SESSION_DIRECTORY
```

`gstat`: monitor the status of submitted jobs

To see the status of all the jobs you have submitted, use the **`gstat`** command. Typing:

```
gstat -s SESSION
```

will print to the screen a table like the following:

```
Job ID    Status
====================
job.12    TERMINATED
job.15    SUBMITTED
job.16    RUNNING
job.17    RUNNING
job.23    NEW
```

Note: If you have never submitted any job, or if you have cleared your job list with the **gclean** command, then **gstat** will print *nothing* to the screen!

A job can be in one and only one of the following states:

`NEW`

The job has been created but not yet submitted: it only exists on the local disk.

`RUNNING`

The job is currently running – there's nothing to do but wait.

`SUBMITTED`

The job has been sent to a compute resource for execution – it should change to `RUNNING` status eventually.

`STOPPED`

The job was sent to a remote cluster for execution, but it is stuck there for some unknown reason. There is no automated procedure in this case: the best thing you can do is to contact the systems administrator to determine what has happened.

`UNKNOWN`

Job info is not found, possibly because the remote resource is currently not accessible due to a network error, a misconfiguration or because the remote resource is not available anymore. When the root cause is fixed, and the resource is available again, the status of the job should automatically move to another state.

`TERMINATED`

The job has finished running; now there are three things you can do:

1. Use the **gget** command to get the command output files back from the remote execution cluster.
2. Use the **gclean** command to remove this job from the list. After issuing **gclean** on a job, any information on it is lost, so be sure you have retrieved any interesting output with **gget** before!
3. If something went wrong during the execution of the job (it did not complete its execution or -possibly- it did not even start), you can use the **ginfo** command to try to debug the problem.

The list of submitted jobs persists from one session to the other: you can log off, shut your computer down, then turn it on again next day and you will see the same list of jobs.

Note: Completed jobs persist in the **gstat** list until they are cleared off with the **gclean** command.

`gtail`: peeking at the job output and error report

Once a job has reached `RUNNING` status (check with **gstat**), you can also monitor its progress by looking at the last lines in the job output and error stream.

An example might clarify this: assume you have submitted a long-running computation as *job.16* and you know from **gstat** that it got into `RUNNING` state; then to take a peek at what this job is doing, you issue the following command:

```
gtail job.16
```

This would produce the following output, from which you can deduce how far GAMESS has progressed into the computation:

```
RECOMMEND NRAD ABOVE  50 FOR ZETA'S ABOVE 1E+4

RECOMMEND NRAD ABOVE  75 FOR ZETA'S ABOVE 1E+5

RECOMMEND NRAD ABOVE 125 FOR ZETA'S ABOVE 1E+6

DFT IS SWITCHED OFF, PERFORMING PURE SCF UNTIL SWOFF THRESHOLD IS REACHED.


ITER EX DEM     TOTAL ENERGY        E CHANGE  DENSITY CHANGE    DIIS ERROR

  1  0  0    -1079.0196780290 -1079.0196780290   0.343816910   1.529879639

         * * *   INITIATING DIIS PROCEDURE   * * *

  2  1  0    -1081.1910665431    -2.1713885141   0.056618918   0.105322104

  3  2  0    -1081.2658345285    -0.0747679855   0.019565324   0.044813607
```

By default, **gtail** only outputs the last 10 lines of a job output/error stream. To see more, use the command line option `-n`; for example, to see the last 25 lines of the output, issue the command:

```
gtail -n 25 job.16
```

The command **gtail** is especially useful for long computations: you can see how far a job has gotten and, e.g., cancel it if it's gotten stuck into an endless/unproductive loop.

To "keep an eye" over what a job is doing, you can add the `-f` option to **gtail**: this will run **gtail** in "follow" mode, i.e., **gtail** will continue to display the contents of the job output and update it as time passes, until you hit Ctrl+C to interrupt it.

gkill: cancel a running job

To cancel a running job, you can use the command **gkill**. For instance, to cancel *job.16*, you would type the following command into the terminal:

```
gkill job.16
```

Warning: *There's no way to undo a cancel operation!* Once you have issued a **gkill** command, the job is deleted and it cannot be resumed. (You can still re-submit it with **gresub**, though.)

gget: retrieve the output of finished jobs

Once a job has reached `RUNNING` status (check with **gstat**), you can retrieve its output files with the **gget** command. For instance, to download the output files of *job.15* you would use:

```
gget job.15
```

This command will print out a message like:

```
Job results successfully retrieved in '/path/to/some/directory'
```

If you are not running the **gget** command on your computer, but rather on a shared front-end like *ocikbgtw*, you can copy+paste the path within quotes to the sftp command to get the files to your usual workstation. For example, you can run the following command in a terminal on your computer to get the output files back to your workstation:

```
sftp ocikbgtw:'/path/to/some/directory'
```

This will take you to the directory where the output files have been stored.

`gclean`: remove a completed job from the status list

Jobs persist in the **`gstat`** list until they are cleared off; you need to use the **`gclean`** command for that.

Just call the **`gclean`** command followed by the job identifier *job.NNN*. For example:

```
gclean job.23
```

In normal operation, you can only remove jobs that are in the `TERMINATED` status; if you want to force **`gclean`** to remove a job that is not in any one of those states, just add `-f` to the command line.

`gresub`: re-submit a failed job

In case a job failed for accidental causes (e.g., the site where it was running went unexpectedly down), you can re-submit it with the **`gresub`** command.

Just call **`gresub`** followed by the job identifier *job.NNN*. For example:

```
gresub job.42
```

Resubmitting a job that is not in a terminal state (i.e., `TERMINATED`) results in the job being killed (as with **`gkill`**) before being submitted again. If you are unsure what state a job is in, check it with **`gstat`**.

`gservers`: list available resources

The **`gservers`** command prints out information about the configured resources. For each resource, a summary of the information recorded in the configuration file and the current resource status is printed. For example:

```
$ gservers
+---------------------------------------------------------------+
|                       smscg                                   |
+===============================================================+
|                   Frontend host name / frontend   giis.smscg.ch |
|                              Access mode / type   arc0          |
|                          Authorization name / auth   smscg      |
|                              Accessible? / updated   1          |
|                    Total number of cores / ncores   4000        |
|                        Total queued jobs / queued   3475        |
|                     Own queued jobs / user_queued   0           |
|                       Own running jobs / user_run   0           |
|           Max cores per job / max_cores_per_job   256           |
| Max memory per core (MB) / max_memory_per_core   2000          |
|  Max walltime per job (minutes) / max_walltime   1440          |
+---------------------------------------------------------------+
```

The meaning of the printed fields is as follows:

- The title of each box is the "resource name", as you would write it after the *-r* option to **`gsub`**.
- *Access mode / type*: it is the kind of software that is used for accessing the resource; consult Section *Configuration File* for more information about resource types.
- *Authorization name / auth*: this is paired with the *Access mode / type*, and identifies a section in the *configuration file* where authentication information for this resource is stored; see Section *Configuration File* for more information.
- *Accessible? / updated*: whether you are *currently* authorized to access this resource; note that if this turns *False* or *0* for resources that you should have access to, then something is wrong either with the state of your system, or with the resource itself. (The procedure on how to diagnose this is too complex to list here; consult your friendly systems administrator :-))
- *Total number of cores*: the total number of cores present on the resource. Note this can vary over time as cluster nodes go in and out of service: computers break, then are repaired, then break again, etc.

- *Total queued jobs*: number of jobs (from all users) waiting to be executed on the remote compute cluster.
- *Own queued jobs*: number of jobs (submitted by you) waiting to be executed on the remote compute cluster.
- *Own running jobs*: number of jobs (submitted by you) currently executing on the remote compute cluster.
- *Max cores per job*: the maximum number of cores that you can request for a single computational job on this resource.
- *Max memory per core*: maximum amount of memory (per core) that you can request on this resource. The amount shows the maximum requestable memory in MB.
- *Max walltime per job*: maximum duration of a computational job on this resource. The amount shows the maximum time in seconds.

The whole point of GC3Utils is to abstract job submission and management from detailed knowledge of the resources and their hardware and software configuration, but it is sometimes convenient and sometimes necessary to get into this level of detail...

`ginfo`: accessing low-level details of a job

It is sometimes necessary, for debugging purposes, to print out all the details about a job; the **`ginfo`** command does just that: prints all the details that GC3Utils know about a single job.

For instance, to print out detailed information about *job.13* in session *TEST1*, you would type:

```
ginfo -s TEST1 job.13
```

For a job in `RUNNING` or `SUBMITTED` state, only little information is known: basically, where the job is running, and when it was started:

```
$ ginfo -s XXX job.13
job.13
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Wed Mar  7 17:40:07 2012
        Submitted to 'smscg' at Wed Mar  7 17:40:07 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/59351331138407177154619 5
    resource_name: smscg
    state_last_changed: 1331138407.33
    timestamp:
        SUBMITTED: 1331138407.33
```

If you omit the job number, information about *all* jobs in the session will be printed.

Most of the output is only useful if you are familiar with GC3Utils inner working. Nonetheless, **`ginfo`** output is definitely something you should include in any report about a misbehaving job!

For a finished job, the information is more complete and can include error messages in case the job has failed:

```
$ ginfo -s TEST1 job.13
job.13
    cores: 1
    download_dir: /home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/exam01
    execution_targets: idgc3grid01.uzh.ch
    log:
        SUBMITTED at Wed Mar  7 15:52:37 2012
        Submitted to 'idgc3grid01' at Wed Mar  7 15:52:37 2012
        TERMINATING at Wed Mar  7 15:54:52 2012
        Final output downloaded to '/home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/e
        TERMINATED at Wed Mar  7 15:54:53 2012
        Execution of gamess terminated normally wed mar  7 15:52:42 2012
    lrms_jobid: gsiftp://idgc3grid01.uzh.ch:2811/jobs/29387133113195716781566 70
    lrms_jobname: exam01
    original_exitcode: 0
```

```
    queue: all.q
    resource_name: idgc3grid01
    state_last_changed: 1331132093.18
    stderr_filename: exam01.out
    stdout_filename: exam01.out
    timestamp:
        SUBMITTED: 1331131957.49
        TERMINATED: 1331132093.18
        TERMINATING: 1331132092.74
    used_cputime: 0
    used_memory: 492019
    used_walltime: 60
```

With option `-v`, **`ginfo`** output is even more verbose and complete, and includes information about the application itself, the input and output files, plus some backend-specific information:

```
$ ginfo -c -s TEST1 job.13
job.13
    application_tag: gamess
    arguments: exam01.inp
    changed: False
    environment:
    executable: /$GAMESS_LOCATION/nggms
    execution:
        _arc0_state_last_checked: 1331138407.33
        _exitcode: None
        _signal: None
        _state: SUBMITTED
        execution_targets: hera.wsl.ch
        log:
            SUBMITTED at Wed Mar  7 17:40:07 2012
            Submitted to 'smscg' at Wed Mar  7 17:40:07 2012
        lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/5935133113840717715461 95
        resource_name: smscg
        state_last_changed: 1331138407.33
        timestamp:
            SUBMITTED: 1331138407.33
    inp_file_path: test/data/exam01.inp
    inputs:
        file:///home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/test/data/exam01.inp:
    job_name: exam01
    jobname: exam01
    join: True
    output_base_url: None
    output_dir: /home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/exam01
    outputs:
        exam01.dat: file, , exam01.dat, None, None, None, None
        exam01.out: file, , exam01.out, None, None, None, None
    persistent_id: job.33998
    requested_architecture: None
    requested_cores: 1
    requested_memory: 2
    requested_walltime: 8
    stderr: None
    stdin: None
    stdout: exam01.out
    tags: APPS/CHEM/GAMESS-2010
    verno: None
```

`gselect`: select job ids from from a session

The **`gselect`** command allows you to select Job IDs from a GC3Pie session that satisfy the selected criteria. This command is usually used in combination with **`gresub`**, **`gkill`**, **`ginfo`**, **`gget`** or **`gclean`**, for instance:

```
$ gselect -l STOPPED | xargs gresub
```

The output of this command is a list of Job IDs, one per line. The criteria specified by command-line options will be AND'ed together, i.e., a job must satisfy all of them in order to be selected.

You can select a job based on the following criteria:

`JobID regexp`

> Use option *–jobid REGEXP* to select jobs whose ID matches the supplied regular expression (case insensitive)

`Job state`

> Use option *–state STATE[,STATE...]* to select jobs in one of the specified states, for instance to select jobs in either STOPPED or SUBMITTED state, run *gselect –state STOPPED,SUBMITTED*.

`exit status`

> You can select jobs that terminated with exit status equal to 0 with *–ok* option. To select failed jobs instead (exit status different from 0), use option *–failed*

`Submission time`

> Use option *–submitted-before DATE* and *–submitted-after DATE* to select jobs submitted before or after a specific date. *DATE* must be in a human readable format recognized by the *parsedatetime <https://pypi.python.org/pypi/parsedatetime/>* module, for instance *in 2 hours*, *yesterday* or *10 November 2014, 1pm*.

`gcloud`: manage VMs created by the EC2 backend

The **`gcloud`** command allows you to show and manage VMs created by the EC2 backend.

To show a list of VMs currently running on the EC2 resources correctly configured run:

```
$ gcloud list
==================================
VMs running on EC2 resource `hobbes`
==================================

+------------+---------+---------------+-------------+--------------+----------+
|     id     |  state  |   public ip   | Nr. of jobs |   image id   | keypair |
+------------+---------+---------------+-------------+--------------+----------+
| i-0000053e | running | 130.60.193.45 |      1      | ami-00000035 | antonio |
+------------+---------+---------------+-------------+--------------+----------+
```

This command will show various information, if available, including the number of jobs currently running (or in *TERMINATED* state) on those VM, so that you can easily identify if there is a VM which is not used by any of yours script and you can safely terminate it.

If you want to terminate a VM run the *gcloud terminate* command. In this case, however, you also have to specify the name of the resource with the option `-r`, and the ID of the VM you want to terminate:

```
$ gcloud terminate -r hobbes i-0000053e
```

An empty output is a signal that the VM has been terminated.

The *EC2* backend keeps track of all the VM it created, so that if a VM is not needed anymore it is able to terminate it automatically. However, sometimes you may need to keep a VM up&running and thus you need to tell the EC2 backend to ignore that VM.

This is possible with the *gcloud forget* command. You must supply the correct resource name with `-r RESOURCE_NAME` and a valid VM ID, and if the command succeeds then the VM will never be used by the EC2 backend. Please note also that after running *gcloud forget*, the VM will not be shown in the output of *gcloud list*:

The following example will explain the behavior:

```
$ gcloud list -r hobbes

====================================
VMs running on EC2 resource `hobbes`
====================================

+------------+---------+---------------+-------------+--------------+---------+
|     id     |  state  |   public ip   | Nr. of jobs |   image id   | keypair |
+------------+---------+---------------+-------------+--------------+---------+
| i-00000540 | pending | 130.60.193.45 |     N/A     | ami-00000035 | antonio |
+------------+---------+---------------+-------------+--------------+---------+
```

then we run *gcloud forget*:

```
$ gcloud forget  -r hobbes i-00000540
```

and we run again *gcloud list*:

```
$ gcloud list -r hobbes

====================================
VMs running on EC2 resource `hobbes`
====================================

  no known VMs are currently running on this resource.
```

You can also create a new VM using the default settings using the *gcloud run* command. In this case too you have to specify the `-r` command line option. The output of this command contains some basic information about the created VM:

```
$ gcloud run -r hobbes
+------------+---------+-----------------------------------------------+-------------+-------------
|     id     |  state  |                   public ip                   | Nr. of jobs |   image id
+------------+---------+-----------------------------------------------+-------------+-------------
| i-00000541 | pending | server-4e68ebc4-ea52-45ff-82d0-79699300b323 |     N/A     | ami-00000035
+------------+---------+-----------------------------------------------+-------------+-------------
```

Please note that while the VM is still in *pending* state, the value of the *public ip* field may be meaningless. A successive run of *gcloud list* should show you the correct *public ip*.

Troubleshooting GC3Pie

This page lists a number of errors and issues that you might run into, together with their solution. Please use the GC3Pie mailing list for further help and for any problem not reported here!

Each section covers a different Python error; the section is named after the error name appearing in the *last line* of the Python traceback. (See section *What is a Python traceback?* below)

Contents

What is a Python traceback?

A *traceback* is a long Python error message, detailing the call stack in the code that lead to a specific error condition.

Tracebacks always look like this one (the number of lines printed, the files involved and the actual error message will, of course, vary):

```
Traceback (most recent call last):
 File "/home/mpackard/gc3pie/bin/gsub", line 9, in <module>
   load_entry_point('gc3pie==1.0rc7', 'console_scripts', 'gsub')()
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/gc3pie-1.0rc7-py2.5.egg/gc3utils/frontend
   import gc3utils.commands
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/gc3pie-1.0rc7-py2.5.egg/gc3utils/commands
   import cli.app
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/pyCLI-2.0.2-py2.5.egg/cli/app.py", line
   from cli.util import ifelse, ismethodof
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/pyCLI-2.0.2-py2.5.egg/cli/util.py", line
   BaseStringIO = StringIO.StringIO
AttributeError: 'module' object has no attribute 'StringIO'
```

Let's analyize how a traceback is formed, top to bottom.

A traceback is *always* started by the line:

```
Traceback (most recent call last):
```

Then follow a number of line pairs like this one:

```
File "/home/mpackard/gc3pie/lib/python2.5/site-packages/gc3pie-1.0rc7-py2.5.egg/gc3utils/frontend
  import gc3utils.commands
```

The first line shows the file name and the line number where the program stopped; the second line displays the instruction that Python was executing when the error occurred. *We shall always omit this part of the traceback in the listings below.*

Finally, the traceback ends with the error message on the *last* line:

```
AttributeError: 'module' object has no attribute 'StringIO'
```

Just look up this error message in the section headers below; if you cannot find any relevant section, please write to the GC3Pie mailing list for help.

Common errors using GC3Pie

This section section lists Python errors that may happen when using GC3Pie; each section is named after the error name appearing in the *last line* of the Python traceback. (See section *What is a Python traceback?* above.)

If you get an error that is not listed here, please get in touch via the GC3Pie mailing list.

AttributeError: *module* object has no attribute *StringIO* This error:

```
Traceback (most recent call last):
 ...
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/pyCLI-2.0.2-py2.5.egg/cli/util.py",
line 28, in <module>
   BaseStringIO = StringIO.StringIO
AttributeError: 'module' object has no attribute 'StringIO'
```

is due to a conflicts of the pyCLI library (prior to version 2.0.3) and the Debian/Ubuntu package *python-stats*

There are three ways to get rid of the error:

1. Uninstall the **python-stats* package* <*python-stats*> (run the command `apt-get remove python-stats` as user `root`)
2. Upgrade pyCLI to version 2.0.3 at least.
3. *Upgrade* GC3Pie, which will force an upgrade of pyCLI.

DistributionNotFound If you get this error:

```
Traceback (most recent call last):
    ...
pkg_resources.DistributionNotFound: gc3pie==1.0rc2
```

It usually means that you didn't run `source ../bin/activate; ./setup.py develop` when upgrading GC3Pie.

Please re-do the steps in the *GC3Pie Upgrade instructions* to fix the error.

ImportError: No module named `pstats` This error only occurs on Debian and Ubuntu GNU/Linux:

```
Traceback (most recent call last):
File ".../pyCLI-2.0.2-py2.6.egg/cli/util.py", line 19, in <module>
   import pstats
ImportError: No module named pstats
```

To solve the issue: install the **python-profiler* package* <*python-profiler*>:

```
apt-get install python-profiler # as `root` user
```

NoResources: Could not initialize any computational resource - please check log and configuration file. This error:

```
Traceback (most recent call last):
  ...
  File ".../src/gc3libs/core.py", line 150, in submit
    raise gc3libs.exceptions.NoResources("Could not initialize any computational resource"
gc3libs.exceptions.NoResources: Could not initialize any computational resource - please check lo
```

can have two different causes:

1. You didn't create a configuration file, or you did not list any resource in it.
2. Some other error prevented the resources from being initialized, or the configuration file from being properly read.

ValueError: I/O operation on closed file Sample error traceback (may be repeated multiple times over):

```
Traceback (most recent call last):
  File "/usr/lib/python2.5/logging/__init__.py", line 750, in emit
    self.stream.write(fs % msg)
ValueError: I/O operation on closed file
```

This is discussed in Issue 182; a fix have been committed to release 1.0, so if you are seeing this error, you are running a pre-release version of GC3Pie and should *Upgrade*.

User-visible changes across releases

This is a list of user-visible changes worth mentioning. In each new release, items are added to the top of the file and identify the version they pertain to.

Contents

GC3Pie 2.4

New features

- The environment variable `GC3PIE_RESOURCE_INIT_ERRORS_ARE_FATAL` can be set to `yes` or `1` to cause GC3Pie to abort if any errors occur while initializing the configured resources. The default behavior of GC3Pie is instead to keep running until there is at least one resource that can be used.
- A resource is now automatically disabled if an unrecoverable error occurs during its use.

GC3Pie 2.3

Incompatible changes

- The ARC backends and supporting code have been removed: it is no longer possible to use GC3Pie to submit tasks to an ARC job manager.

- The environment variable `GC3PIE_NO_CATCH_ERRORS` now can specify a list of patterns to selectively unignore unexpected/generic errors in the code. As this feature should only be used in debugging code, we allow ourselves to break backwards compatibility.
- The cloud and mathematics libraries are no longer installed by default with `pip install gc3pie` – please use:

```
pip install gc3pie[openstack,ec2,optimizer]
```

  to install support for all optional backends and libraries.
- The *gc3libs.utils.ifelse* function was removed in favor of Python's ternary operator.

New features

- New task collection `DependentTaskCollection` to run a collection of tasks with given pre/post dependencies across them.
- GC3Pie will now parse and obey the `Port`, `Identity`, `User`, `ConnectionTimeout`, and `ProxyCommand` options from the SSH config file. Location of an alternate configuration file to use with GC3Pie can be set in any `[auth/*]` section of type SSH; see the *Configuration File* reference for details. Thanks to Niko Eherenfeuchter and Karandash8 for feature requests and preliminary implementations.
- Application prologue and epilogue scripts can now be embedded in the GC3Pie configuration file, or referenced by file name.
- New selection options have been added to the gselect command.
- *gc3libs.Configuration* will now raise different exceptions depending on whether no files could be read (`NoAccessibleConfigurationFile`) or could not be parsed (`NoValidConfigurationFile`).

Important bug fixes

- Shell metacharacters are now allowed in *Application* arguments. Each argument string is now properly quoted before passing it to the execution layer.
- LSF backend updated to work with *both* bjobs *and* bacct for accounting, or to parse information provided in the final output file as a last resort.
- All backends should now set a Task's *returncode* and *exitcode* values according to the documented meaning. Thanks to Y. Yakimovitch for reporting the issue.

GC3Pie 2.2

New features

- New `openstack` backend for running jobs on ephemeral VMs on OpenStack-compatible IaaS cloud systems. This is preferred over the OpenStack EC2 compatibility layer.
- New configurable scheduler for GC3Pie's `Engine`
- Session-based scripts can now snapshot the output of RUNNING jobs at every cycle.
- ARC backends are now deprecated: they will be removed in the next major version of GC3Pie.
- The `pbs` backend can now handle also Altair's PBSPro.

Changes to command-line utilities

- `gget`: New option *-A* to download output files of *all* tasks in a session.
- `gget`: New option `-c`/`--changed-only` to only download files that have apparently changed remotely.
- The GC3Apps collection has been enriched with several new applications.

Important bug fixes

- Working directory for remote jobs using the `shellcmd` backend is now stored in `/var/tmp` instead of `/tmp`, which should allow results to be retrieved even after a reboot of the remote machine.

GC3Pie 2.1

New features and incompatible changes

- GC3Pie now requires Python 2.6 or above to run.
- New `ec2` backend for running jobs on ephemeral VMs on EC2-compatible IaaS cloud systems.
- New package *gc3libs.optimizer* to find local optima of functions that can be computed through a job. Currently only implements the "Differential Evolution" algorithm, but the framework is generic enough to plug any genetic algorithm.
- New configuration options `prolog_content` and `epilog_content`, to allow execute oneliners before or after the command without having to create an auxiliary file.
- New `resourcedir` option for `shellcmd` resources. This is used to modify the default value for the directory containing job informations.

Changes to command-line utilities

- New command `gcloud` to interface with cloud-based VMs that were spawned by GC3Pie to run jobs.
- Table output now uses a different formatting (we use Python's *prettytable* package instead of the *texttable* package that we were using before, due to Py3 compatibility).

GC3Pie 2.0

New features and incompatible changes

- GC3Pie can now run on MacOSX.
- A session now has a configurable storage location, which can be a directory on the filesystem (FilesystemStore, the default so far) or can be a table in an SQL database (of any kind supported by SQLAlchemy).
- New ARC1 backend to use ARC resources through the new NorduGrid 1.x library API.
- New backend "subprocess": execute applications as local processes.
- New backends for running on various batch-queueing systems: SLURM, LSF, PBS.
- Implement recursive upload and download of directories if they are specified in an *Application*'s *input* or *output* attribute.
- New execution state *TERMINATING*: task objects are in this state when execution is finished remotely, but the task output has not yet been retrieved.
- Reorganize documentation and move it to http://gc3pie.readthedocs.org/
- Script logging is now controlled by a single configuration file *.gc3/gc3utils.log.conf*
- Session-based scripts now print WARNING messages to STDERR by default (previously, only ERROR messages were logged).
- Add caching to ARC backends, to reduce the number of network queries.
- Use GNU ".~NUMBER~" format for backup directories.

Configuration file changes

- Rename ARC0 resource type to *arc0*

Changes to command-line utilities

- New *gsession* command to manage sessions.
- The `glist` command was renamed to `gservers`
- The `gsub` and `gnotify` commands were removed.
- The `PATH` tag no longer gets any special treatment in session-based scripts `--output` processing.
- `ginfo`: New option `--tabular` to print information in table format.
- `gkill`: New option *-A*/*–all* to remove all jobs in a session.
- Use the *rungms* script to execute GAMESS.

API changes

- Module `gc3libs.dag` has been renamed to `gc3libs.workflow`.
- API changes in *gc3libs.cmdline.SessionBasedScript* allow *new_tasks()* in *SessionBasedScript* instances to return *Task* instances instead of quadruples.
- Interpret *Application.requested_memory* as the *total* memory for the job.
- the `Resource` and `LRMS` objects were merged
- the `gc3libs.scheduler` module has been removed; its functionality is now incorporated in the `Application` class.
- configuration-related code moved into *gc3libs.config* module
- removed the application registry.
- New package *gc3libs.compat* to provide 3rd-party functionality that is not present in all supported versions of Python.
- Implement *gc3libs.ANY_OUTPUT* to retrieve the full contents of the output directory, whatever it is.
- New *RetryableTask* class to wrap a task and re-submit it on failure until some specified condition is met.

GC3Pie 1.0

Configuration file changes

- Renamed configuration file to `gc3pie.conf`: the file `gc3utils.conf` will no longer be read!
- SGE clusters must now have `type = sge` in the configuration file (instead of `type = ssh-sge`)
- All computational resource must have an `architecture = ...` line; see the ConfigurationFile wiki page for details
- Probably more changes than it's worth to list here: check your configuration against the **configuration_** page!

Command-line utilities changes

- GC3Utils and GC3Apps (`grosetta`/`ggamess`/etc.) now all accept a `-s`/`--session` option for locating the job storage directory: this allows grouping jobs into folders instead of shoveling them all into `~/.gc3/jobs`.
- GC3Apps: replaced option `-t`/`--table` with `-l`/`--states`. The new option prints a table of submitted jobs in addition to the summary stats; if a comma-separated list of job states follows the option, only job in those states are printed.
- Command `gstat` will now print a summary of the job states if the list is too long to fit on screen; use the -v option to get the full job listing regardless of its length.

- Command `gstat` can now print information on jobs in a certain state only; see help text for option `--state`
- Removed `-l` option from ginfo; use `-v` instead.
- GC3Utils: all commands accepting multiple job IDs on the command line, now exit with the number of errors/failures occurred. Since exit codes are practically limited to 7 bits, exit code 126 means that more than 125 failures happened.

GC3Pie 0.10

- First release for public use outside of GC3

2.2 Programmer Documentation

This document is the technical reference for the GC3Libs programming model, aimed at programmers who want to use GC3Libs to implement computational workflows in Python.

The *Programming overview* section is the starting point for whoever wants to start developing applications with GC3Pie. It gives an overview of the main components of the library and how they interact with each other.

The Tutorials section contains documentation that describes in more detail the various components discussed in the programming overview, as well as many working examples (took from exercises done during the training events) and the *Warholize Tutorial*: a step-by-step tutorial that will show you how to write a complex GC3Pie workflow.

The *GC3Libs programming API* section instead contains the API reference of GC3Pie library.

2.2.1 Programming overview

Computational job lifecycle

A computational job (for short: *job*) is a single run of a non-interactive application. The prototypical example is a run of GAMESS on a single input file.

The GC3Utils commands support the following workflow:

1. Submit a GAMESS job (with a single input file): **`ggamess`**
2. Monitor the status of the submitted job: **`gstat`**
3. Retrieve the output of a job once it's finished: **`gget`**

Usage and some examples on how to use the mentioned commands are provided in the next sections

Managing jobs with GC3Libs

GC3Libs takes an application-oriented approach to asynchronous computing. A generic `Application` class provides the basic operations for controlling remote computations and fetching a result; client code should derive specialized sub-classes to deal with a particular application, and to perform any application-specific pre- and post-processing.

The generic procedure for performing computations with GC3Libs is the following:

1. Client code creates an instance of an *Application* sub-class.
2. Asynchronous computation is started by submitting the application object; this associates the application with an actual (possibly remote) computational job.
3. Client code can monitor the state of the computational job; state handlers are called on the application object as the state changes.

4. When the job is done, the final output is retrieved and a post-processing method is invoked on the application object.

At this point, results of the computation are available and can be used by the calling program.

The `Application` class (and its sub-classes) alow client code to control the above process by:

1. Specifying the characteristics (computer program to run, input/output files, memory/CPU/duration requirements, etc.) of the corresponding computational job. This is done by passing suitable values to the `Application` constructor. See the `Application` constructor documentation for a detailed description of the parameters.
2. Providing methods to control the "life-cycle" of the associated computational job: start, check execution state, stop, retrieve a snapshot of the output files. There are actually two different interfaces for this, detailed below:

   (a) A *passive* interface: a `Core` or a `Engine` object is used to start/stop/monitor jobs associated with the given application. For instance:

```
a = GamessApplication(...)

# create a `Core` object; only one instance is needed
g = Core(...)

# start the remote computation
g.submit(a)

# periodically monitor job execution
g.update_job_state(a)

# retrieve output when the job is done
g.fetch_output(a)
```

   The passive interface gives client code full control over the lifecycle of the job, but cannot support some use cases (e.g., automatic application re-start).

   As you can see from the above example, the passive interface is implemented by methods in the `Core` and `Engine` classes (they implement the same interface). See those classes documentation for more details.

   (b) An *active* interface: this requires that the `Application` object be attached to a `Core` or `Engine` instance:

```
a = GamessApplication(...)

# create a `Core` object; only one instance is needed
g = Core(...)

# tell application to use the active interface
a.attach(g)

# start the remote computation
a.submit()

# periodically monitor job execution
a.update_job_state()

# retrieve output when the job is done
a.fetch_output()
```

   With the active interface, application objects can support automated restart and similar use-cases.

   When an `Engine` object is used instead of a `Core` one, the job life-cycle is automatically managed, providing a fully asynchronous way of executing computations.

The active interface is implemented by the `Task` class and all its descendants (including `Application`).

3. Providing "state transition methods" that are called when a change in the job execution state is detected; those methods can implement application specific behavior, like restarting the computational job with changed input if the alloted duration has expired but the computation has not finished. In particular, a *postprocess* method is called when the final output of an application is available locally for processing.

   The set of "state transition methods" currently implemented by the `Application` class are: `new()`, `submitted()`, `running()`, `stopped()`, `terminated()` and `postprocess()`. Each method is called when the execution state of an application object changes to the corresponding state; see each method's documentation for exact information.

In addition, GC3Libs provides *collection* classes, that expose interfaces *2.* and *3.* above, allowing one to control a set of applications as a single whole. Collections can be nested (i.e., a collection can hold a mix of `Application` and `TaskCollection` objects), so that workflows can be implemented by composing collection objects.

Note that the term *computational job* (or just *job*, for short) is used here in a quite general sense, to mean any kind of computation that can happen independently of the main thread of the calling program. GC3Libs currently provide means to execute a job as a separate process on the same computer, or as a batch job on a remote computational cluster.

Execution model of GC3Libs applications

An *Application* can be regarded as an abstraction of an independent asynchronous computation, i.e., a GC3Libs' *Application* behaves much like an independent UNIX process (but it can actually run on a separate remote computer). Indeed, GC3Libs' *Application* objects mimic the POSIX process model: *Application* are started by a parent process, run independently of it, and need to have their final exit code and output reaped by the calling process.

The following table makes the correspondence between POSIX processes and GC3Libs' *Application* objects explicit.

os module function	*Core* function	purpose
exec	Core.submit	start new job
kill(..., SIGTERM)	Core.kill	terminate executing job
wait(..., WNOHANG)	Core.update_job_state	get job status
•	Core.fetch_output	retrieve output

Note:

1. With GC3Libs, it is not possible to send an arbitrary signal to a running job: jobs can only be started and stopped (killed).
2. Since POSIX processes are always executed on the local machine, there is no equivalent of the GC3Libs *fetch_output*.

Application exit codes

POSIX encodes process termination information in the "return code", which can be parsed through *os.WEXITSTATUS*, *os.WIFSIGNALED*, *os.WTERMSIG* and relative library calls.

Likewise, GC3Libs provides each `Application` object with an *execution.returncode* attribute, which is a valid POSIX "return code". Client code can therefore use *os.WEXITSTATUS* and relatives to inspect it; convenience attributes *execution.signal* and *execution.exitcode* are available for direct access to the parts of the return code. See `Run.returncode()` for more information.

However, GC3Libs has to deal with error conditions that are not catered for by the POSIX process model: for instance, execution of an application may fail because of an error connecting to the remote execution cluster.

To this purpose, GC3Libs encodes information about abnormal job termination using a set of pseudo-signal codes in a job's *execution.returncode* attribute: i.e., if termination of a job is due to some grid/batch system/middleware error, the job's *os.WIFSIGNALED(app.execution.returncode)* will be *True* and the signal code (as gotten from *os.WTERMSIG(app.execution.returncode)*) will be one of those listed in the `Run.Signals` documentation.

Application execution states

At any given moment, a GC3Libs job is in any one of a set of pre-defined states, listed in the table below. The job state is always available in the *.execution.state* instance property of any *Application* or *Task* object; see `Run.state()` for detailed information.

GC3Libs' Job state	purpose	can change to
NEW	Job has not yet been submitted/started (i.e., gsub not called)	SUBMITTED (by gsub)
SUBMITTED	Job has been sent to execution resource	RUNNING, STOPPED
STOPPED	Trap state: job needs manual intervention (either user- or sysadmin-level) to resume normal execution	TERMINATED (by gkill), SUBMITTED (by miracle)
RUNNING	Job is executing on remote resource	TERMINATED
UNKNOWN	Job info not found or lost track of job (e.g., network error or invalid job ID)	any other state
TERMINATED	Job execution is finished (correctly or not) and will not be resumed	None: final state

When an `Application` object is first created, its *.execution.state* attribute is assigned the state NEW. After a successful start (via *Core.submit()* or similar), it is transitioned to state SUBMITTED. Further transitions to RUNNING or STOPPED or TERMINATED state, happen completely independently of the creator program: the *Core.update_job_state()* call provides updates on the status of a job. (Somewhat like the POSIX *wait(..., WNOHANG)* system call, except that GC3Libs provide explicit RUNNING and STOPPED states, instead of encoding them into the return value.)

The STOPPED state is a kind of generic "run time error" state: a job can get into the STOPPED state if its execution is stopped (e.g., a SIGSTOP is sent to the remote process) or delayed indefinitely (e.g., the remote batch system puts the job "on hold"). There is no way a job can get out of the STOPPED state automatically: all transitions from the STOPPED state require manual intervention, either by the submitting user (e.g., cancel the job), or by the remote systems administrator (e.g., by releasing the hold).

The UNKNOWN state is a temporary error state: whenever GC3Pie is unable to get any information on the job, its state move to UNKNOWN. It is usually related to a (hopefully temporary) failure while accessing the remote resource, because of a network error or because the resource is not correctly configured. After the underlying cause of the error is fixed and GC3Pie is able again to get information on the job, its state will change to the proper state.

The TERMINATED state is the final state of a job: once a job reaches it, it cannot get back to any other state. Jobs reach TERMINATED state regardless of their exit code, or even if a system failure occurred during remote execution; actually, jobs can reach the TERMINATED status even if they didn't run at all!

A job that is not in the NEW or TERMINATED state is said to be a "live" job.

Computational job specification

One of the purposes of GC3Libs is to provide an abstraction layer that frees client code from dealing with the details of job execution on a possibly remote cluster. For this to work, it necessary to specify job characteristics and requirements, so that the GC3Libs scheduler can select an appropriate computational resource for executing the job.

GC3Libs *Application* provide a way to describe computational job characteristics (program to run, input and output files, memory/duration requirements, etc.) loosely patterned after ARC's xRSL language.

The description of the computational job is done through keyword parameters to the `Application` constructor, which see for details. Changes in the job characteristics *after* an `Application` object has been constructed are not currently supported.

UML Diagram

An UML diagram of GC3Pie classes is available (also in PNG format)

2.2.2 GC3Pie programming tutorials

Slides on specific topics

This is a list of tutorials made for the GC3Pie 2012 Training event that has been held in Zurich on 1st and 2nd of October 2012. The slides and tutorials are an introduction to the GC3Pie python package.

Introduction to GC3Pie

> Introduction to the software: what is GC3Pie, what is it for, and an overview of its features for writing high-throughput computing scripts.

Basic GC3Pie programming

> The *Application* class, the smallest building block of GC3Pie. Introduction to the concept of Job, states of an application and to the *Core* class.

Application requirements

> How to define extra requirements for an application, such as the minimum amount of memory it will use, the number of cores needed or the architecture of the CPUs.

Managing applications: the SessionBasedScript class

> Introduction to the highest-level interface to build applications with GC3Pie, the *SessionBasedScript*. Information on how to create simple scripts that take care of the execution of your applications, from submission to getting back the final results.

The GC3Utils commands

> Low-level tools to aid debugging the scripts.

Introduction to Workflows with GC3Pie

> Using a practical example (the *Warholize Tutorial*) we show how workflows are implemented with GC3Pie. The following slides will cover in more details the single steps needed to produce a complex workflow.

ParallelTaskCollection

> Description of the *ParallelTaskCollection* class, used to run tasks in parallel.

StagedTaskCollection

> Description of the *StagedTaskCollection* class, used to run a sequence of a fixed number of jobs.

SequentialTaskCollection

> Description of the *SequentialTaskCollection* class, used to run a sequence of jobs that can be altered during runtime.

Example scripts

gdemo_simple.py

> Simplest script you can create. It only uses *Application* and *Engine* classes to create an application, submit it, check its status and retrieve its output.

grun.py

a *SessionBasedScript* that executes its argument as command. It can also run it multiple times by wrapping it in a ParallelTaskCollection or a SequentialTaskCollection, depending on a command line option. Useful for testing a configured resource.

gdemo_session.py

a simple *SessionBasedScript* that sums two values by customizing a *SequentialTaskCollection*.

warholize.py

an enhanched version of the *warholize* script proposed in the *Warholize Tutorial*

Warholize Tutorial

In this tutorial we will show you how to use GC3Pie libraries in order to build a command line script which will run a complex workflow with both parallel and sequential tasks.

The tutorial itself contains the complete source code of the application (cfr. Literate Programming on Wikipedia), so that you will be able to test/modify it and produce a working `warholize.py` script by downloading the `pylit.py`:file: script from the PyLit Homepage and running the following command on the `gc3pie/docs/programmers/tutorials/warholize/warholize.txt` file, from within the SVN tree of GC3Pie:

```
$ ./pylit warholize.txt warholize.py
```

Introduction

Warholize is a GC3Pie demo application to produce, from a generic image picture, a new picture like the famous Warhol's work: Marylin. The script uses the powerful ImageMagick set of tools (at least version 6.3.5-7). This tutorial will assume that both *ImageMagick* and *GC3Pie* are already installed and configured.

In order to produce a similar image we have to do a series of transformations on the picture:

1. convert the original image to grayscale.
2. *colorize* the grayscale image using three different colors each time, based on the gray levels. We may, for instance, make all pixels with luminosity between 0-33% in red, pixels between 34-66% in yellow and pixels between 67% and 100% in green.

   To do that, we first have to:

   (a) create a *Color Lookup Table* (LUT) using a combination of three randomly chosen colors

   (b) apply the LUT to the grayscale image
3. Finally, we can merge together all the colorized images and produce our *warholized* image.

Clearly, step 2) depends on the step 1), and 3) depends on 2), so we basically have a sequence of tasks, but since step 2) need to create *N* different independent images, we can parallelize this step.

From top to bottom

We will write our script starting from the top and will descend to the bottom, from the command line script, to the workflow and finally to the single execution units which compose the application.

The script

The *SessionBasedScript* class in the *gc3libs.cmdline* module is used to create a generic script. It already have all what is needed to read gc3pie configuration files, manage resources, schedule jobs etc. The only missing thing is, well, your application!

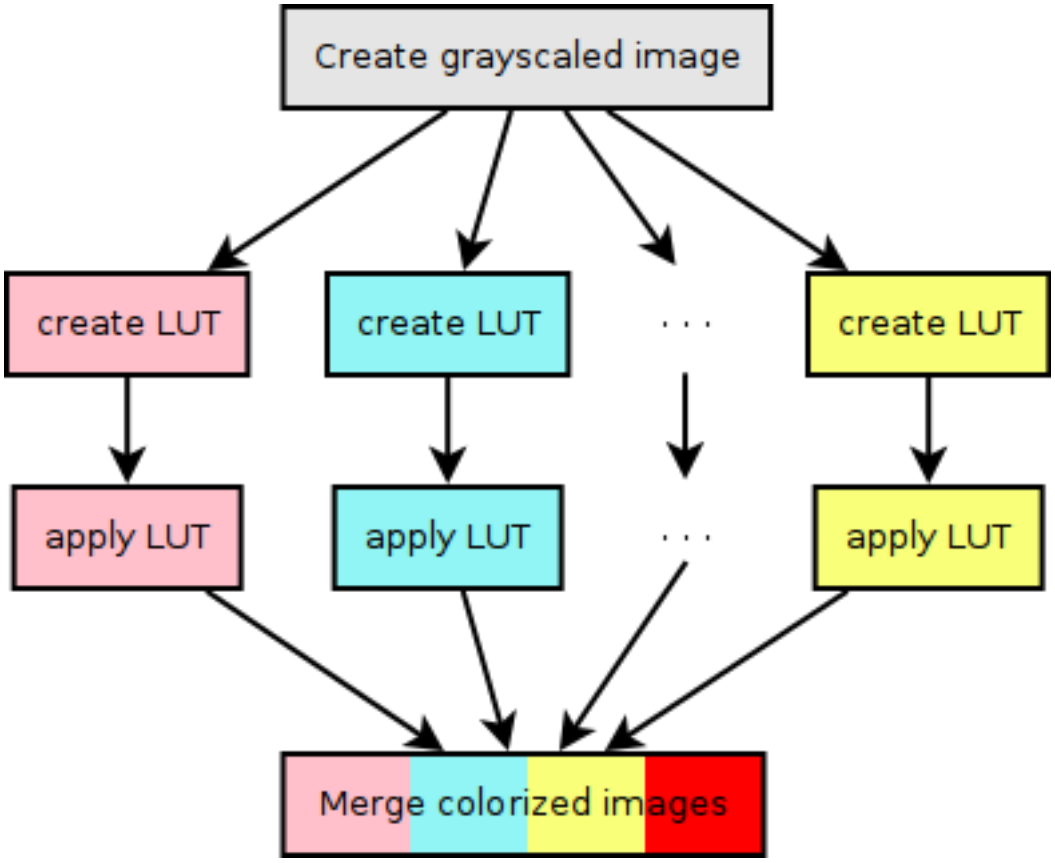

Fig. 2.1: Workflow of the *warholize* script

Let's start by creating a new empty file and importing some basic modules:

```
import os
import gc3libs
from gc3libs.cmdline import SessionBasedScript
```

we then create a class which inherits from *SessionBasedScript* (in GC3Pie, most of the customizations are done by inheriting from a more generic class and overriding the *__init__* method and possibly others):

```
class WarholizeScript(SessionBasedScript):
    """
    Demo script to create a `Warholized` version of an image.
    """
    version='1.0'
```

Please note that you must either write a small docstring, or add a *description* attribute. These values are used when the script is called with options `--help` or `--version`, which are automatically added by GC3Pie.

The way we want to use our script is straightforward:

```
$  warholize.py inputfile [inputfiles ...]
```

and this will create a directory `Warholized.<inputfile>` in which there will be a file called `warhol_<inputfile>` containing the desired warholized image (and a lot of temporary files, at least for now).

But we may want to add some additional options to the script, in order to decide how many colorized pictures the warholized image will be made of, or if we want to resize the image. *SessionBasedScript* uses the PyCLI module which is, in turn, a wrapper around standard argparse (or optparse for older pythons) module. To customize the script you may define a *setup_options* method and put in there some calls to *SessionBasedScript.add_param()*, which is inherited from *cli.app.CommandLineApp*:

```
def setup_options(self):
    self.add_param('--copies', default=4, type=int,
                   help="Number of copyes (Default:4). It has to be a perfect square!")
```

In this example we will accept a `--copies` option to define how many colorized copies the final picture will be made of. Please refer to the documentation of the PyCLI module for details on the syntax of the *add_param* method.

The *heart* of the script is, however, the *new_tasks* method, which will be called to create the initial tasks of the scripts. In our case it will be something like:

```
def new_tasks(self, extra):
    gc3libs.log.info("Creating main sequential task")
    for (i, input_file) in enumerate(self.params.args):
        extra_args = extra.copy()
        extra_args['output_dir'] = 'Warholized.%s' % os.path.basename(input_file)
        yield WarholizeWorkflow(input_file,
                                self.params.copies,
                                **extra_args)
```

new_tasks is used as a *generator* (but it could return a list as well). Each *yielded* object is a *task*. In GC3Pie, a *task* is either a single application or a complex workflow, and rapresents an *execution unit*. In our case we create a *WarholizeWorkflow* task which is the workflow described before.

In our case we yield a different *WarholizeWorkflow* task for each input file. These tasks will run in parallel.

Please note that we are using the *gc3libs.log* module to log informations about the execution. This module works like the logging module and has methods like *error*, *warning*, *info* or *debug*, but its logging level is automatically configured by *SessionBasedScript*'s constructor. This way you can increase the verbosity of your script by simply adding `-v` options from the command line.

The workflows

Main sequential workflow The module *gc3libs.workflow* contains two main objects: *SequentialTaskCollection* and *ParallelTaskCollection*. They execute tasks in serial and in parallel, respectively. We will use both of them to create our workflow; the first one, *WarholizeWorkflow*, is a sequential task, therefore we have to inherit from *SequentialTaskCollection* and customize its *__init__* method:

```
from gc3libs.workflow import SequentialTaskCollection, ParallelTaskCollection
import math
from gc3libs import Run

class WarholizeWorkflow(SequentialTaskCollection):
    """
    Main workflow.
    """

    def __init__(self, input_image,  copies, **extra_args):
        self.input_image = input_image
        self.output_image = "warhol_%s" % os.path.basename(input_image)

        gc3libs.log.info(
            "Producing a warholized version of input file %s "
            "and store it in %s" % (input_image, self.output_image))

        self.output_dir = os.path.relpath(extra_args.get('output_dir'))

        self.copies = copies

        # Check that copies is a perfect square
        if math.sqrt(self.copies) != int(math.sqrt(self.copies)):
            raise gc3libs.exceptions.InvalidArgument(
```

```
                "`copies` argument must be a perfect square.")

        self.jobname = extra_args.get('jobname', 'WarholizedWorkflow')
        self.grayscaled_image = "grayscaled_%s" % os.path.basename(self.input_image)
```

Up to now we just parsed the arguments. The following lines, instead, create the first task that we want to execute. By now, we can create only the first one, *GrayScaleConvertApplication*, which will produce a grayscale image from the input file:

```
self.tasks = [
    GrayScaleConvertApplication(
        self.input_image, self.grayscaled_image, self.output_dir,
        self.output_dir),
    ]
```

Finally, we call the parent's constructor.:

```
SequentialTaskCollection.__init__(
    self, self.tasks)
```

This will create the initial task list, but we have to run also step 2 and 3, and this is done by creating a *next* method. This method will be called after all the tasks in *self.tasks* are finished. We cannot create all the jobs at once because we don't have all the needed input files yet. Please note that by creating the tasks in the *next* method you could decide *at runtime* which tasks to run next and what arguments we may want to give to them.

In our case, however, the *next* method is quite simple:

```
def next(self, iteration):
    last = self.tasks[-1]

    if iteration == 0:
        # first time we got called.  We have the grayscaled image,
        # we have to run the Tricolorize task.
        self.add(TricolorizeMultipleImages(
            os.path.join(self.output_dir, self.grayscaled_image),
            self.copies, self.output_dir))
        return Run.State.RUNNING
    elif iteration == 1:
        # second time, we already have the colorized images, we
        # have to merge them together.
        self.add(MergeImagesApplication(
            os.path.join(self.output_dir, self.grayscaled_image),
            last.warhol_dir,
            self.output_image))
        return Run.State.RUNNING
    else:
        self.execution.returncode = last.execution.returncode
        return Run.State.TERMINATED
```

At each iteration, we call *self.add()* to add an instance of a task-like class (*gc3libs.Application*, *gc3libs.workflow.ParallelTaskCollection* or *gc3libs.workflow.SequentialTaskCollection*, in our case) to complete the next step, and we return the current state, which will be *gc3libs.Run.State.RUNNING* unless we have finished the computation.

Step one: convert to grayscale *GrayScaleConvertApplication* is the application responsible to convert to grayscale the input image. The command we want to execute is:

```
$    convert -colorspace gray <input_image> grayscaled_<input_image>
```

To create a generic application we create a class which inherit from *gc3libs.Application* and we usually only need to customize the *__init__* method:

```
# An useful function to copy files
from gc3libs.utils import copyfile

class GrayScaleConvertApplication(gc3libs.Application):
    def __init__(self, input_image, grayscaled_image, output_dir, warhol_dir):
        self.warhol_dir = warhol_dir
        self.grayscaled_image = grayscaled_image

        arguments = [
            'convert',
            os.path.basename(input_image),
            '-colorspace',
            'gray',
            ]

        gc3libs.log.info(
            "Craeting  GrayScale convert application from file %s"
            "to file %s" % (input_image, grayscaled_image))

        gc3libs.Application.__init__(
            self,
            arguments = arguments + [grayscaled_image],
            inputs = [input_image],
            outputs = [grayscaled_image, 'stderr.txt', 'stdout.txt'],
            output_dir = output_dir,
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

Creating a *gc3libs.Application* is straigthforward: you just call the constructor with the executable, the arguments, and the input/output files you will need.

If you don't specify the `output_dir` directory, gc3pie libraries will create one starting from the class name. If the output directory exists already, the old one will be renamed.

To do any kind of post processing you can define a *terminate* method for your application. It will be called after your application will terminate. In our case we want to copy the gray scale version of the image to the *warhol_dir*, so that it will be easily reachable by all other applications:

```
def terminated(self):
    """Move grayscale image to the main output dir"""
    copyfile(
        os.path.join(self.output_dir, self.grayscaled_image),
        self.warhol_dir)
```

Step two: parallel workflow to create colorized images

The *TricolorizeMultipleImages* is responsible to create multiple versions of the grayscale image with different coloration chosen randomly from a list of available colors. It does it by running multiple instance of *TricolorizeImage* with different arguments. Since we want to run the various colorization in parallel, it inherits from *gc3libs.workflow.ParallelTaskCollection* class. Like we did for *GrayScaleConvertApplication*, we only need to customize the constructor *__init__*, creating the various subtasks we want to run:

```
import itertools
import random

class TricolorizeMultipleImages(ParallelTaskCollection):
    colors = ['yellow', 'blue', 'red', 'pink', 'orchid',
              'indigo', 'navy', 'turquoise1', 'SeaGreen', 'gold',
              'orange', 'magenta']

    def __init__(self, grayscaled_image, copies, output_dir):
```

```
        gc3libs.log.info(
            "TricolorizeMultipleImages for %d copies run" % copies)
        self.jobname = "Warholizer_Parallel"
        ncolors = 3
        ### XXX Why I have to use basename???
        self.output_dir = os.path.join(
            os.path.basename(output_dir), 'tricolorize')
        self.warhol_dir = output_dir

        # Compute a unique sequence of random combination of
        # colors. Please note that we can have a maximum of N!/3! if N
        # is len(colors)
        assert copies <= math.factorial(len(self.colors)) / math.factorial(ncolors)

        combinations = [i for i in itertools.combinations(self.colors, ncolors)]
        combinations = random.sample(combinations, copies)

        # Create all the single tasks
        self.tasks = []
        for i, colors in enumerate(combinations):
            self.tasks.append(TricolorizeImage(
                os.path.relpath(grayscaled_image),
                "%s.%d" % (self.output_dir, i),
                "%s.%d" % (grayscaled_image, i),
                colors,
                self.warhol_dir))

        ParallelTaskCollection.__init__(self, self.tasks)
```

The main loop will fill the *self.tasks* list with various *TricolorizedImage* tasks, each one with an unique combination of three colors to use to generate the colorized image. The GC3Pie framework will then run these tasks in parallel, on any available resource.

The *TricolorizedImage* class is indeed a *SequentialTaskCollection*, since it has to generate the LUT first, and then apply it to the grayscale image. We already saw how to create a *SequentialTaskCollection*: we modify the constructor in order to add the first job (*CreateLutApplication*), and the *next* method will take care of running the *ApplyLutApplication* application on the output of the first job:

```
class TricolorizeImage(SequentialTaskCollection):
    """
    Sequential workflow to produce a `tricolorized` version of a
    grayscale image
    """
    def __init__(self, grayscaled_image, output_dir, output_file,
                 colors, warhol_dir):
        self.grayscaled_image = grayscaled_image
        self.output_dir = output_dir
        self.warhol_dir = warhol_dir
        self.jobname = 'TricolorizeImage'
        self.output_file = output_file

        if not os.path.isdir(output_dir):
            os.mkdir(output_dir)

        gc3libs.log.info(
            "Tricolorize image %s to %s" % (
                self.grayscaled_image, self.output_file))

        self.tasks = [
            CreateLutApplication(
                self.grayscaled_image,
                "%s.miff" % self.grayscaled_image,
                self.output_dir,
```

```
                colors, self.warhol_dir),
            ]

        SequentialTaskCollection.__init__(self, self.tasks)

    def next(self, iteration):
        last = self.tasks[-1]
        if iteration == 0:
            # First time we got called. The LUT has been created, we
            # have to apply it to the grayscale image
            self.add(ApplyLutApplication(
                self.grayscaled_image,
                os.path.join(last.output_dir, last.lutfile),
                os.path.basename(self.output_file),
                self.output_dir, self.warhol_dir))
            return Run.State.RUNNING
        else:
            self.execution.returncode = last.execution.returncode
            return Run.State.TERMINATED
```

The *CreateLutApplication* is again an application which inherits from *gc3libs.Application*. The command we want to execute is something like:

```
$    convert -size 1x1 xc:<color1> xc:<color2> xc:<color3> +append -resize 256x1! <output_file.mi
```

This will basically create an image 256x1 pixels big, made of a gradient using all the listed colors. The code will look like:

```
class CreateLutApplication(gc3libs.Application):
    """Create the LUT for the image using 3 colors picked randomly
    from CreateLutApplication.colors"""

    def __init__(self, input_image, output_file, output_dir, colors, working_dir):
        self.lutfile = os.path.basename(output_file)
        self.working_dir = working_dir
        gc3libs.log.info("Creating lut file %s from %s using "
                         "colors: %s" % (
            self.lutfile, input_image, str.join(", ", colors)))
        gc3libs.Application.__init__(
            self,
            arguments = [
                'convert',
                '-size',
                '1x1'] + [
                "xc:%s" % color for color in colors] + [
                '+append',
                '-resize',
                '256x1!',
                self.lutfile,
                ],
            inputs = [input_image],
            outputs = [self.lutfile, 'stdout.txt', 'stderr.txt'],
            output_dir = output_dir + '.createlut',
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

Similarly, the *ApplyLutApplication* application will run the following command:

```
$    convert grayscaled_<input_image> <lutfile.N.miff> -clut grayscaled_<input_image>.<N>
```

This command will apply the LUT to the grayscaled image: it will modify the grayscaled image by *coloring* a generic pixel with a luminosity value of *n* (which will be an integer value from 0 to 255, of course) with the color

at position *n* in the LUT image (actually, *n+1*). Each *ApplyLutApplication* will save the resulting image to a file named as `grayscaled_<input_image>.<N>`.

The class will look like:

```
class ApplyLutApplication(gc3libs.Application):
    """Apply the LUT computed by `CreateLutApplication` to
    `image_file`"""

    def __init__(self, input_image, lutfile, output_file, output_dir, working_dir):

        gc3libs.log.info("Applying lut file %s to %s" % (lutfile, input_image))
        self.working_dir = working_dir
        self.output_file = output_file

        gc3libs.Application.__init__(
            self,
            arguments = [
                'convert',
                os.path.basename(input_image),
                os.path.basename(lutfile),
                '-clut',
                output_file,
                ],
            inputs = [input_image, lutfile],
            outputs = [output_file, 'stdout.txt', 'stderr.txt'],
            output_dir = output_dir + '.applylut',
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

The *terminated* method:

```
def terminated(self):
    """Copy colorized image to the output dir"""
    copyfile(
        os.path.join(self.output_dir, self.output_file),
        self.working_dir)
```

will copy the colorized image file in the top level directory, so that it will be easier for the last application to find all the needed files.

Step three: merge all them together At this point we will have in the main output directory a bunch of files named after `grayscaled_<input_image>.N` with N a sequential integer and `<input_image>` the name of the original image. The last application, *MergeImagesApplication*, will produce a `warhol_<input_image>` image by merging all of them using the command:

```
$    montage grayscaled_<input_image>.* -tile 3x3 -geometry +5+5 -background white warholized_<inp
```

Now it should be easy to write such application:

```
import re

class MergeImagesApplication(gc3libs.Application):
    def __init__(self, grayscaled_image, input_dir, output_file):
        ifile_regexp = re.compile(
            "%s.[0-9]+" % os.path.basename(grayscaled_image))
        input_files = [
            os.path.join(input_dir, fname) for fname in os.listdir(input_dir)
            if ifile_regexp.match(fname)]
        input_filenames = [os.path.basename(i) for i in input_files]
        gc3libs.log.info("MergeImages initialized")
        self.input_dir = input_dir
```

```
        self.output_file = output_file

        tile = math.sqrt(len(input_files))
        if tile != int(tile):
            gc3libs.log.error(
                "We would expect to have a perfect square"
                "of images to merge, but we have %d instead" % len(input_files))
            raise gc3libs.exceptions.InvalidArgument(
                "We would expect to have a perfect square of images to merge, but we have %d inst

        gc3libs.Application.__init__(
            self,
            arguments = ['montage'] + input_filenames + [
                '-tile',
                '%dx%d' % (tile, tile),
                '-geometry',
                '+5+5',
                '-background',
                'white',
                output_file,
                ],
            inputs = input_files,
            outputs = [output_file, 'stderr.txt', 'stdout.txt'],
            output_dir = os.path.join(input_dir, 'output'),
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

Making the script executable

Finally, in order to make the script *executable*, we add the following lines to the end of the file. The *WarholizeScritp().run()* call will be executed only when the file is run as a script, and will do all the magic related to argument parsing, creating the session etc...:

```
if __name__ == '__main__':
    import warholize
    warholize.WarholizeScript().run()
```

Please note that the `import warholize` statement is important to address issue 95 and make the gc3pie scripts work with your current session (*gstat*, *ginfo*...)

Testing

To test this script I would suggest to use the famous *Lena* picture, which can be found in the *miscelaneous* section of the Signal and Image Processing Institute page. Download the image, rename it as `lena.tiff` and run the following command:

```
$    ./warholize.py -C 1 lena.tiff --copies 9
```

(add `-r localhost` if your gc3pie.conf script support it and you want to test it locally).

After completion a file `Warholized.lena.tiff/output/warhol_lena.tiff` will be created.

2.2.3 GC3Libs programming API

gc3libs

GC3Libs is a python package for controlling the life-cycle of a Grid or batch computational job.

Fig. 2.2: Warholized version of *Lena*

GC3Libs provides services for submitting computational jobs to Grids and batch systems, controlling their execution, persisting job information, and retrieving the final output.

GC3Libs takes an application-oriented approach to batch computing. A generic `Application` class provides the basic operations for controlling remote computations, but different `Application` subclasses can expose adapted interfaces, focusing on the most relevant aspects of the application being represented.

class gc3libs.**Application**(*arguments*, *inputs*, *outputs*, *output_dir*, ***extra_args*)

Support for running a generic application with the GC3Libs. The following parameters are *required* to create an *Application* instance:

arguments List or sequence of program arguments. The program to execute is the first one.; any object in the list will be converted to string via Python's *str()*.

inputs Files that will be copied to the remote execution node before execution starts.

There are two possible ways of specifying the *inputs* parameter:

- It can be a Python dictionary: keys are local file paths or URLs, values are remote file names.
- It can be a Python list: each item in the list should be a pair *(source, remote_file_name)*: the *source* can be a local file or a URL; *remote_file_name* is the path (relative to the execution directory) where *source* will be downloaded. If *remote_file_name* is an absolute path, an `InvalidArgument` error is raised.

  A single string *file_name* is allowed instead of the pair and results in the local file *file_name* being copied to *file_name* on the remote host.

outputs Files and directories that will be copied from the remote execution node back to the local computer (or a network-accessible server) after execution has completed. Directories are copied recursively.

There are three possible ways of specifying the *outputs* parameter:

- It can be a Python dictionary: keys are remote file or directory paths (relative to the execution directory), values are corresponding local names.
- It can be a Python list: each item in the list should be a pair *(remote_file_name, destination)*: the *destination* can be a local file or a URL; *remote_file_name* is the path (relative to the execution directory) that will be uploaded to *destination*. If *remote_file_name* is an absolute path, an `InvalidArgument` error is raised.

  A single string *file_name* is allowed instead of the pair and results in the remote file *file_name* being copied to *file_name* on the local host.
- The constant *gc3libs.ANY_OUTPUT* which instructs GC3Libs to copy every file in the remote execution directory back to the local output path (as specified by the *output_dir* attribute).

Note that no errors will be raised if an output file is not present. Override the `terminated()` method to raise errors for reacting on this kind of failures.

output_dir Path to the base directory where output files will be downloaded. Output file names are interpreted relative to this base directory.

***requested_cores*,'requested_memory','requested_walltime'** specify resource requirements for the application: * the number of independent execution units (CPU cores; all are

required to be in the same execution node),

- amount of memory (as a `gc3libs.quantity.Memory` object),
- amount of wall-clock time to allocate for the computational job (as a `gc3libs.quantity.Duration` object).

The following optional parameters may be additionally specified as keyword arguments and will be given special treatment by the *Application* class logic:

requested_architecture specify that this application can only be executed on a certain processor architecture; see `Run.Arch` for a list of possible values. The default value *None* means that any architecture is valid, i.e., there are no requirements on the processor architecture.

environment a dictionary defining environment variables and the values to give them in the task execution setting. Keys of the dictionary are environmental variables names, and dictionary values define the corresponding variable content. Both keys and values must be strings or convertible to string.

For example, to run the application in an environment where the variable `LC_ALL` has the value `C` and the variable `HZ` has the value `100`, one would use:

```
Application(...,
  environment={'LC_ALL':'C', 'HZ':100},
  ...)
```

output_base_url if not *None*, this is prefixed to all output files (except stdout and stderr, which are always retrieved), so, for instance, having *output_base_url="gsiftp://example.org/data"* will upload output files into that remote directory.

stdin file name of a file whose contents will be fed as standard input stream to the remote-executing process.

stdout name of a file where the standard output stream of the remote executing process will be redirected to; will be automatically added to *outputs*.

stderr name of a file where the standard error stream of the remote executing process will be redirected to; will be automatically added to *outputs*.

join if this evaluates to *True*, then standard error is redirected to the file specified by *stdout* and *stderr* is ignored. (*join* has no effect if *stdout* is not given.)

tags list of tag names (string) that must be present on a resource in order to be eligible for submission.

Any other keyword arguments will be set as instance attributes, but otherwise ignored by the *Application* constructor.

After successful construction, an *Application* object is guaranteed to have the following instance attributes:

arguments list of strings specifying command-line arguments for executable invocation. The first element must be the executable.

inputs dictionary mapping source URL (a `gc3libs.url.Url` object) to a remote file name (a string); remote file names are relative paths (root directory is the remote job folder)

outputs dictionary mapping remote file name (a string) to a destination (a `gc3libs.url.Url`); remote file names are relative paths (root directory is the remote job folder)

output_dir Path to the base directory where output files will be downloaded. Output file names (those which are not URLs) are interpreted relative to this base directory.

execution

a *Run* instance; its state attribute is initially set to `NEW` (Actually inherited from the `Task`)

environment dictionary mapping environment variable names to the requested value (string); possibly empty

stdin *None* or a string specifying a (local) file name. If *stdin* is not None, then it matches a key name in *inputs*

stdout *None* or a string specifying a (remote) file name. If *stdout* is not None, then it matches a key name in *outputs*

stderr *None* or a string specifying a (remote) file name. If *stdout* is not None, then it matches a key name in *outputs*

join boolean value, indicating whether *stdout* and *stderr* are collected into the same file

tags list of strings specifying the tags to request in each resource for submission; possibly empty.

application_name = 'generic'
A name for applications of this class.

This string is used as a prefix for configuration items related to this application in configured resources. For example, if the *application_name* is `foo`, then the application interface code in GC3Pie might search for `foo_cmd`, `foo_extra_args`, etc. See *qsub_sge()* for an actual example.

bsub(*resource*, *_suppress_warning=False*, ***extra_args*)
Get an LSF `qsub` command-line invocation for submitting an instance of this application. Return a pair *(cmd_argv, app_argv)*, where *cmd_argv* is a list containing the *argv*-vector of the command to run to submit an instance of this application to the LSF batch system, and *app_argv* is the *argv*-vector to use when invoking the application.

In the construction of the command-line invocation, one should assume that all the input files (as named in *Application.inputs*) have been copied to the current working directory, and that output files should be created in this same directory.

The default implementation just prefixes any output from the *cmdline* method with an LSF `bsub` invocation of the form `bsub -cwd . -L /bin/sh` + resource limits.

Override this method in application-specific classes to provide appropriate invocation templates and/or add resource-specific submission options.

cmdline(*resource*)
Return list of command-line arguments for invoking the application.

This is exactly the *argv*-vector of the application process: the application command name is included as first item (index 0) of the list, further items are command-line arguments.

Hence, to get a UNIX shell command-line, just concatenate the elements of the list, separating them with spaces.

compatible_resources(*resources*)
Return a list of compatible resources.

fetch_output(*download_dir*, *overwrite*, *changed_only*, ***extra_args*)
Call the corresponding method of the controller.

fetch_output_error(*ex*)
Invocation of *Core.fetch_output()* on this object failed; *ex* is the *Exception* that describes the error.

If this method returns an exception object, that is raised as a result of the *Core.fetch_output()*, otherwise the return value is ignored and *Core.fetch_output* returns *None*.

Default is to return *ex* unchanged; override in derived classes to change this behavior.

qsub_pbs(*resource*, *_suppress_warning=False*, ***extra_args*)
Similar to *qsub_sge()*, but for the PBS/TORQUE resource manager.

qsub_sge(*resource*, ***extra_args*)
Get an SGE `qsub` command-line invocation for submitting an instance of this application.

Return a pair *(cmd_argv, app_argv)*. Both *cmd_argv* and *app_argv* are *argv*-lists: the command name is included as first item (index 0) of the list, further items are command-line arguments; *cmd_argv* is the *argv*-list for the submission command (excluding the actual application command part); *app_argv* is the *argv*-list for invoking the application. By overriding this method, one can add futher resource-specific options at the end of the *cmd_argv argv*-list.

In the construction of the command-line invocation, one should assume that all the input files (as named in *Application.inputs*) have been copied to the current working directory, and that output files should be created in this same directory.

The default implementation just prefixes any output from the *cmdline* method with an SGE `qsub` invocation of the form `qsub -cwd -S /bin/sh` + resource limits. Note that *there is no generic way of requesting a certain number of cores* in SGE: it all depends on the installed parallel environment, and these are totally under control of the local sysadmin; therefore, any request for cores is ignored and a warning is logged.

Override this method in application-specific classes to provide appropriate invocation templates and/or add different submission options.

rank_resources(*resources*)
Sort the given resources in order of preference.

By default, computational resource *a* is preferred over *b* if it has less queued jobs from the same user; failing that, if it has more free slots; failing that, if it has less queued jobs (in total); finally, should all preceding parameters compare equal, *a* is preferred over *b* if it has less running jobs from the same user.

Resources where the job has already attempted to run (the resource front-end name is recorded in *.execution._execution_targets*) are then moved to the back of the list, to avoid resubmitting to a faulty resource.

sbatch(*resource*, ***extra_args*)
Get a SLURM `sbatch` command-line invocation for submitting an instance of this application.

Return a pair *(cmd_argv, app_argv)*. Both *cmd_argv* and *app_argv* are *argv*-lists: the command name is included as first item (index 0) of the list, further items are command-line arguments; *cmd_argv* is the *argv*-list for the submission command (excluding the actual application command part); *app_argv* is the *argv*-list for invoking the application. By overriding this method, one can add futher resource-specific options at the end of the *cmd_argv argv*-list.

In the construction of the command-line invocation, one should assume that all the input files (as named in *Application.inputs*) have been copied to the current working directory, and that output files should be created in this same directory.

Override this method in application-specific classes to provide appropriate invocation templates and/or add different submission options.

submit_error(*exs*)
Invocation of *Core.submit()* on this object failed; *exs* is a list of *Exception* objects, one for each attempted submission.

If this method returns an exception object, that is raised as a result of the *Core.submit()*, otherwise the return value is ignored and *Core.submit* returns *None*.

Default is to always return the first exception in the list (on the assumption that it is the root of all exceptions or that at least it refers to the preferred resource). Override in derived classes to change this behavior.

update_job_state_error(*ex*)
Handle exceptions that occurred during a *Core.update_job_state* call.

If this method returns an exception object, that exception is processed in *Core.update_job_state()* instead of the original one. Any other return value is ignored and *Core.update_job_state* proceeds as if no exception had happened.

Argument *ex* is the exception that was raised by the backend during job state update.

Default is to return *ex* unchanged; override in derived classes to change this behavior.

class gc3libs.**Default**
A namespace for all constants and default values used in the GC3Libs package.

class gc3libs.**Run**(*initializer=None*, *attach=None*, ***keywd*)
A specialized *dict*-like object that keeps information about the execution state of an *Application* instance.

A *Run* object is guaranteed to have the following attributes:

log A *gc3libs.utils.History* instance, recording human-readable text messages on events in this job's history.

info A simplified interface for reading/writing messages to *Run.log*. Reading from the *info* attribute returns the last message appended to *log*. Writing into *info* appends a message to *log*.

timestamp Dictionary, recording the most recent timestamp when a certain state was reached. Timestamps are given as UNIX epochs.

For properties *state*, *signal* and *returncode*, see the respective documentation.

Run objects support attribute lookup by both the `[...]` and the `.` syntax; see *gc3libs.utils.Struct* for examples.

class `Arch`

Processor architectures, for use as values in the *requested_architecture* field of the *Application* class constructor.

The following values are currently defined:

X86_64 64-bit Intel/AMD/VIA x86 processors in 64-bit mode.

X86_32 32-bit Intel/AMD/VIA x86 processors in 32-bit mode.

`Run.`**`exitcode`**

The "exit code" part of a *Run.returncode*, see *os.WEXITSTATUS*. This is an 8-bit integer, whose meaning is entirely application-specific. (However, the value *255* is often used to mean that an error has occurred and the application could not end its execution normally.)

`Run.`**`in_state`**`(*names)`

Return *True* if the *Run* state matches any of the given names.

In addition to the states from `Run.State`, the two additional names `ok` and `failed` are also accepted, with the following meaning:

- `ok`: state is *TERMINATED* and *returncode* is 0.
- `failed`: state is *TERMINATED* and *returncode* is non-zero.

`Run.`**`info`**

A simplified interface for reading/writing entries into *history*.

Setting the *info* attribute appends a message to the log:

```
>>> j1 = Run()
>>> j1.info = 'a message'
>>> j1.info = 'a second message'
```

Getting the value of the *info* attribute returns the last message entered in the log:

```
>>> j1.info
u'a second message ...'
```

`Run.`**`returncode`**

The *returncode* attribute of this job object encodes the *Run* termination status in a manner compatible with the POSIX termination status as implemented by *os.WIFSIGNALED* and *os.WIFEXITED*.

However, in contrast with POSIX usage, the *exitcode* and the *signal* part can *both* be significant: in case a Grid middleware error happened *after* the application has successfully completed its execution. In other words, *os.WEXITSTATUS(returncode)* is meaningful iff *os.WTERMSIG(returncode)* is 0 or one of the pseudo-signals listed in *Run.Signals*.

Run.exitcode and *Run.signal* are combined to form the return code 16-bit integer as follows (the convention appears to be obeyed on every known system):

Bit	Encodes...
0..7	signal number
8	1 if program dumped core.
9..16	exit code

Note: the "core dump bit" is always 0 here.

Setting the *returncode* property sets *exitcode* and *signal*; you can either assign a *(signal, exitcode)* pair to *returncode*, or set *returncode* to an integer from which the correct *exitcode* and *signal* attribute values are extracted:

```
>>> j = Run()
>>> j.returncode = (42, 56)
>>> j.signal
42
>>> j.exitcode
56

>>> j.returncode = 137
>>> j.signal
9
>>> j.exitcode
0
```

See also *Run.exitcode* and *Run.signal*.

static Run.**shellexit_to_returncode**(*rc*)

Convert a shell exit code to a POSIX process return code.

A POSIX shell represents the return code of the last-run program within its exit code as follows:

- If the program was terminated by signal *N*, the shell exits with code 128+N,
- otherwise, if the program terminated with exit code C, the shell exits with code C.

Run.**signal**

The "signal number" part of a *Run.returncode*, see *os.WTERMSIG* for details.

The "signal number" is a 7-bit integer value in the range 0..127; value *0* is used to mean that no signal has been received during the application runtime (i.e., the application terminated by calling `exit()`).

The value represents either a real UNIX system signal, or a "fake" one that GC3Libs uses to represent Grid middleware errors (see *Run.Signals*).

Run.**state**

The state a *Run* is in.

The value of *Run.state* must always be a value from the *Run.State* enumeration, i.e., one of the following values.

+————————+————————————————————————————+————————
———-+ # noqa |Run.State value|purpose |can change to | # noqa
+================+===+====
noqa |NEW |Job has not yet been submitted/started (i.e., gsub not called)|SUBMITTED (by
gsub) | # noqa +————————+————————————————————————————+————————
—-+ # noqa |SUBMITTED |Job has been sent to execution resource |RUNNING, STOPPED
| # noqa +————————+————————————————————————————+————————-+ #
noqa |STOPPED |Trap state: job needs manual intervention (either user- |TERMINATING(by
gkill),| # noqa | |or sysadmin-level) to resume normal execution |SUBMITTED (by miracle)| #
noqa +————————+————————————————————————————+————————-+ # noqa
|RUNNING |Job is executing on remote resource |TERMINATING | # noqa +————————+——
————————————————————————————+————————-+ # noqa |TERMINATING |Job
has finished execution on remote resource; |TERMINATED | # noqa | |output not yet retrieved | | #
noqa +————————+————————————————————————————+————————-+ # noqa
|TERMINATED |Job execution is finished (correctly or not) |None: final state | # noqa | |and will not
be resumed; output has been retrieved | | # noqa +————————+————————————————————
————————+————————-+ # noqa

When a *Run* object is first created, it is assigned the state NEW. After a successful invocation of *Core.submit()*, it is transitioned to state SUBMITTED. Further transitions to RUNNING or STOPPED or TERMINATED state, happen completely independently of the creator progra; the *Core.update_job_state()* call provides updates on the status of a job.

The STOPPED state is a kind of generic "run time error" state: a job can get into the STOPPED state if its execution is stopped (e.g., a SIGSTOP is sent to the remote process) or delayed indefinitely (e.g., the remote batch system puts the job "on hold"). There is no way a job can get out of the STOPPED

state automatically: all transitions from the STOPPED state require manual intervention, either by the submitting user (e.g., cancel the job), or by the remote systems administrator (e.g., by releasing the hold).

The TERMINATED state is the final state of a job: once a job reaches it, it cannot get back to any other state. Jobs reach TERMINATED state regardless of their exit code, or even if a system failure occurred during remote execution; actually, jobs can reach the TERMINATED status even if they didn't run at all, for example, in case of a fatal failure during the submission step.

class gc3libs.**Task**(***extra_args*)

Mix-in class implementing a facade for job control.

A *Task* can be described as an "active" job, in the sense that all job control is done through methods on the *Task* instance itself; contrast this with operating on *Application* objects through a *Core* or *Engine* instance.

The following pseudo-code is an example of the usage of the *Task* interface for controlling a job. Assume that *GamessApplication* is inheriting from *Task* (it actually is):

```
t = GamessApplication(input_file)
t.submit()
# ... do other stuff
t.update_state()
# ... take decisions based on t.execution.state
t.wait() # blocks until task is terminated
```

Each *Task* object has an *execution* attribute: it is an instance of class *Run*, initialized with a new instance of *Run*, and at any given time it reflects the current status of the associated remote job. In particular, *execution.state* can be checked for the current task status.

After successful initialization, a *Task* instance will have the following attributes:

changed evaluates to *True* if the *Task* has been changed since last time it has been saved to persistent storage (see gclibs.persistence)

execution a *Run* instance; its state attribute is initially set to NEW.

attach(*controller*)

Use the given Grid interface for operations on the job associated with this task.

detach()

Remove any reference to the current grid interface. After this, calling any method other than *attach()* results in an exception TaskDetachedFromGridError being thrown.

fetch_output(*output_dir=None*, *overwrite=False*, *changed_only=True*, ***extra_args*)

Retrieve the outputs of the computational job associated with this task into directory *output_dir*, or, if that is *None*, into the directory whose path is stored in instance attribute *.output_dir*.

If the execution state is *TERMINATING*, transition the state to *TERMINATED* (which runs the appropriate hook).

See gc3libs.Core.fetch_output() for a full explanation.

Returns Path to the directory where the job output has been collected.

free(***extra_args*)

Release any remote resources associated with this task.

See gc3libs.Core.free() for a full explanation.

kill(***extra_args*)

Terminate the computational job associated with this task.

See gc3libs.Core.kill() for a full explanation.

new()

Called when the job state is (re)set to *NEW*.

Note this will not be called when the application object is created, rather if the state is reset to *NEW* after it has already been submitted.

The default implementation does nothing, override in derived classes to implement additional behavior.

peek(*what='stdout'*, *offset=0*, *size=None*, ***extra_args*)
Download *size* bytes (at offset *offset* from the start) from the associated job standard output or error stream, and write them into a local file. Return a file-like object from which the downloaded contents can be read.

See `gc3libs.Core.peek()` for a full explanation.

progress()
Advance the associated job through all states of a regular lifecycle. In detail:

1. If *execution.state* is *NEW*, the associated job is started.
2. The state is updated until it reaches *TERMINATED*
3. Output is collected and the final returncode is returned.

An exception *TaskError* is raised if the job hits state *STOPPED* or *UNKNOWN* during an update in phase 2.

When the job reaches *TERMINATING* state, the output is retrieved; if this operation is successfull, state is advanced to *TERMINATED*.

Once the job reaches *TERMINATED* state, the return code (stored also in *.returncode*) is returned; if the job is not yet in *TERMINATED* state, calling *progress* returns *None*.

Raises exception `UnexpectedStateError` if the associated job goes into state *STOPPED* or *UNKNOWN*

Returns final returncode, or *None* if the execution state is not *TERMINATED*.

running()
Called when the job state transitions to *RUNNING*, i.e., the job has been successfully started on a (possibly) remote resource.

The default implementation does nothing, override in derived classes to implement additional behavior.

stopped()
Called when the job state transitions to *STOPPED*, i.e., the job has been remotely suspended for an unknown reason and cannot automatically resume execution.

The default implementation does nothing, override in derived classes to implement additional behavior.

submit(*resubmit=False*, *targets=None*, ***extra_args*)
Start the computational job associated with this *Task* instance.

submitted()
Called when the job state transitions to *SUBMITTED*, i.e., the job has been successfully sent to a (possibly) remote execution resource and is now waiting to be scheduled.

The default implementation does nothing, override in derived classes to implement additional behavior.

terminated()
Called when the job state transitions to *TERMINATED*, i.e., the job has finished execution (with whatever exit status, see *returncode*) and the final output has been retrieved.

The location where the final output has been stored is available in attribute *self.output_dir*.

The default implementation does nothing, override in derived classes to implement additional behavior.

terminating()
Called when the job state transitions to *TERMINATING*, i.e., the remote job has finished execution (with whatever exit status, see *returncode*) but output has not yet been retrieved.

The default implementation does nothing, override in derived classes to implement additional behavior.

unknown()
Called when the job state transitions to *UNKNOWN*, i.e., the job has not been updated for a certain period of time thus it is placed in UNKNOWN state.

Two possible ways of changing from this state: 1) next update cycle, job status is updated from the remote server 2) derive this method for Application specific logic to deal with this case

The default implementation does nothing, override in derived classes to implement additional behavior.

update_state(***extra_args*)

In-place update of the execution state of the computational job associated with this *Task*. After successful completion, *.execution.state* will contain the new state.

After the job has reached the *TERMINATING* state, the following attributes are also set:

execution.duration Time lapse from start to end of the job at the remote execution site, as a *gc3libs.quantity.Duration* value. (This is also often referred to as the 'wall-clock time' or *walltime* of the job.)

execution.max_used_memory Maximum amount of RAM used during job execution, represented as a *gc3libs.quantity.Memory* value.

execution.used_cpu_time Total time (as a *gc3libs.quantity.Duration* value) that the processors has been actively executing the job's code.

The execution backend may set additional attributes; the exact name and format of these additional attributes is backend-specific. However, you can easily identify the backend-specific attributes because their name is prefixed with the (lowercased) backend name; for instance, the `PbsLrms` backend sets attributes *pbs_queue*, *pbs_end_time*, etc.

wait(*interval=60*)

Block until the associated job has reached *TERMINATED* state, then return the job's return code. Note that this does not automatically fetch the output.

Parameters **interval** (*integer*) – Poll job state every this number of seconds

gc3libs.**configure_logger**(*level=40*, *name=None*, *format='sphinx-build: [%(asctime)s] %(levelname)-8s: %(message)s'*, *datefmt='%Y-%m-%d %H:%M:%S'*)

Configure the `gc3.gc3libs` logger.

Arguments *level*, *format* and *datefmt* set the corresponding arguments in the *logging.basicConfig()* call.

If a user configuration file exists in file NAME.log.conf in the `Default.RCDIR` directory (usually `~/.gc3`), it is read and used for more advanced configuration; if it does not exist, then a sample one is created.

gc3libs.**create_engine**(**conf_files*, ***extra_args*)

Returns a *gc3libs.core.Engine* class.

It accepts an optional list of configuration filenames. If the filenames contain a ~ or a variable name, it will be expanded automatically.

Called without arguments, a configuration file will be searched in ~/.gc3/gc3pie.conf and used, if found .

gc3libs.**error_ignored**(**ctx*)

Return `True` if no object in list *ctx* matches the contents of the `GC3PIE_NO_CATCH_ERRORS` environment variable.

Note that the list of un-ignored errors is determined when the *gc3libs* module is initially loaded and is thus insensitive to changes in the environment that happen afterwards.

The calling interface is so designed, that a list of keywords describing -or related- to the error are passed; if any of them has been mentioned in the environment variable `GC3PIE_NO_CATCH_ERRORS` then this function returns `False` – i.e., the error is never ignored by GC3Pie and always propagated to the top-level handler.

gc3libs.application

Support for running a generic application with the GC3Libs.

gc3libs.application.apppot

Support for AppPot-hosted applications.

For more details about AppPot, visit: <http://apppot.googlecode.com>

class gc3libs.application.apppot.**AppPotApplication**(*arguments*, *inputs*, *outputs*, *output_dir*, *apppot_img=None*, *apppot_changes=None*, *apppot_tag='ENV/APPPOT-0.21'*, *apppot_extra=[]*, ***extra_args*)

Base class for AppPot-hosted applications. Provides the same interface as the base `Application` and runs the specified command in an AppPot instance.

In addition to the standard `Application` keyword arguments, the following ones can be given to steer the AppPot execution:

- *apppot_img*: Path or URL to the AppPot system image to use. If `None` (default), then the default AppPot system image on the remote system is used.
- *apppot_changes*: Path or URL to an AppPot changes file to be merged at system startup.
- *apppot_tag*: ARC RTE to use for submission of this AppPot job.
- *apppot_extra*: List of additional UML boot command-line arguments. (Passed to the AppPot instance via `apppot-start`'s `--extra` option.)

gc3libs.application.codeml

Simple interface to the CODEML application.

class gc3libs.application.codeml.**CodemlApplication**(**ctls*, ***extra_args*)

Run a CODEML job with the specified '.ctl' files.

The given '.ctl' input files are parsed and the '.phy' and '.nwk' files mentioned therein are added to the list of files to be copied to the execution site.

static aux_files(*ctl_path*)

Return full path to the seqfile and treefile referenced in the '.ctl' file given as arguments.

terminated()

Set the exit code of a *CodemlApplication* job by inspecting its `.mlc` output files.

An output file is valid iff its last line of each output file reads `Time used: MM:SS` *or* `Time used: HH:MM:SS`

The exit status of the whole job is a bit field composed as follows:

bit no.	meaning
0	H1.mlc valid (0=valid, 1=invalid)
1	H1.mlc present (0=present, 1=no file)
2	H0.mlc valid (0=valid, 1=invalid)
3	H0.mlc present (0=present, 1=not present)
7	error running codeml (1=error, 0=ok)

The special value 127 is returned in case `codeml` did not run at all (Grid or remote cluster error).

So, exit code 0 means that all files processed successfully, code 1 means that `H0.mlc` has not been downloaded (for whatever reason).

TODO:

- Check if the stderr is empty.

gc3libs.application.demo

Specialized support for computational jobs running simple demo.

class gc3libs.application.demo.**Square**(*x*)

Square class, takes a filename containing a list of integer to be squared. writes an output containing the square of each of them

gc3libs.application.gamess

Specialized support for computational jobs running GAMESS-US.

class gc3libs.application.gamess.**GamessAppPotApplication**(*inp_file_path*, **other_input_files*, ***extra_args*)

Specialized *AppPotApplication* object to submit computational jobs running GAMESS-US.

This class makes no check or guarantee that a GAMESS-US executable will be available in the executing AppPot instance: the *apppot_img* and *apppot_tag* keyword arguments can be used to select the AppPot system image to run this application; see the `AppPotApplication` for information.

The *__init__* construction interface is compatible with the one used in *GamessApplication*. The only required parameter for construction is the input file name; any other argument names an additional input file, that is added to the *Application.inputs* list, but not otherwise treated specially.

Any other keyword parameter that is valid in the *Application* class can be used here as well, with the exception of *input* and *output*. Note that a GAMESS-US job is *always* submitted with *join* = *True*, therefore any *stderr* setting is ignored.

class gc3libs.application.gamess.**GamessApplication**(*inp_file_path*, **other_input_files*, ***extra_args*)

Specialized *Application* object to submit computational jobs running GAMESS-US.

The only required parameter for construction is the input file name; subsequent positional arguments are additional input files, that are added to the *Application.inputs* list, but not otherwise treated specially.

The *verno* parameter is used to request a specific version of GAMESS-US; if the default value `None` is used, the default version of GAMESS-US at the executing site is run.

Any other keyword parameter that is valid in the *Application* class can be used here as well, with the exception of *input* and *output*. Note that a GAMESS-US job is *always* submitted with *join* = *True*, therefore any *stderr* setting is ignored.

terminated()

Append to log the termination status line as extracted from the GAMESS '.out' file.

The job exit code *.execution.exitcode* is (re)set according to the following table:

Exit code	Meaning
0	the output file contains the string `EXECUTION OF GAMESS TERMINATED normally`
1	the output file contains the string `EXECUTION OF GAMESS TERMINATED -ABNORMALLY-`
2	the output file contains the string `ddikick exited unexpectedly`
70 (*os.EX_SOFTWARE*)	the output file cannot be read or does not match any of the above patterns

gc3libs.application.rosetta

Specialized support for computational jobs running programs in the Rosetta suite.

class gc3libs.application.rosetta.**RosettaApplication**(*application*, *application_release*, *inputs*, *outputs=[]*, *flags_file=None*, *database=None*, *arguments=[]*, ***extra_args*)

Specialized *Application* object to submit one run of a single application in the Rosetta suite.

Required parameters for construction:

- *application*: name of the Rosetta application to call (e.g., "docking_protocol" or "relax")
- *inputs*: a *dict* instance, keys are Rosetta `-in:file:*` options, values are the (local) path names of the corresponding files. (Example: `inputs={"-in:file:s":"1brs.pdb"}`)
- *outputs*: list of output file names to fetch after Rosetta has finished running.

Optional parameters:

- *flags_file*: path to a local file containing additional flags for controlling Rosetta invocation; if *None*, a local configuration file will be used.
- *database*: (local) path to the Rosetta DB; if this is not specified, then it is assumed that the correct location will be available at the remote execution site as environment variable `ROSETTA_DB_LOCATION`
- *arguments*: If present, they will be appended to the Rosetta application command line.

terminated()

Extract output files from the tar archive created by the 'rosetta.sh' script.

class gc3libs.application.rosetta.**RosettaDockingApplication**(*pdb_file_path*, *native_file_path=None*, *number_of_decoys_to_create=1*, *flags_file=None*, *application_release='3.1'*, ***extra_args*)

Specialized *Application* class for executing a single run of the Rosetta "docking_protocol" application.

Currently used in the *gdocking* app.

gc3libs.application.turbomole

Specialized support for TurboMol.

class gc3libs.application.turbomole.**TurbomoleApplication**(*program*, *control*, **others*, ***extra_args*)

Run TURBOMOLE's *program* on the given *control* file. Any additional arguments are considered additional filenames to input files (e.g., the `coord` file) and copied to the execution directory.

Parameters

- **program** (*str*) – Name of the TURBOMOLE's program to run (e.g., `ridft`)
- **control** (*str*) – Path to a file in TURBOMOLE's `control` format.
- **others** – Path(s) to additional input files.

class gc3libs.application.turbomole.**TurbomoleDefineApplication**(*program*, *define_in*, *coord*, **others*, ***extra_args*)

Run TURBOMOLE's 'define' with the given *define_in* file as input, then run *program* on the *control* file produced.

Any additional arguments are considered additional filenames to input files and copied to the execution directory.

Parameters

- **program** (*str*) – Name of the TURBOMOLE's program to run (e.g., `ridft`)
- **define_in** (*str*) – Path to a file containing keystrokes to pass

as input to the 'define' program.

Parameters **coord** (*str*) – Path to a file containing the molecule

coordinates in TURBOMOLE's format.

Parameters **others** – Path(s) to additional input files.

gc3libs.authentication

Authentication support for the GC3Libs.

class `gc3libs.authentication.`**Auth**(*config*, *auto_enable*)

A mish-mash of authorization functions.

This class actually serves the purposes of:

- a registry of authorization 'types', mapping internally-assigned names to Python classes;
- storage for the configuration information (which can be arbitrary, but should probably be read off a configuration file);
- a factory, returning a 'SomeAuth' object through which clients can deal with actual authorization issues (like checking if the authorization credentials are valid and getting/renewing them).
- a cache, that tries to avoid expensive re-initializations of *Auth* objects by allowing only one live instance per type, and returning it when requested.

FIXME

There are several problems with this approach:

- the configuration is assumed *static* and cannot be changed after the *Auth* instance is constructed.
- there is no communication between the client class and the *Auth* classes.
- there is no control over the lifetime of the cache; at a minimum, it should be settable per-auth-type.
- I'm unsure whether the mapping of 'type names' (as in the *type=...* keyword in the config file) to Python classes belongs in a generic factory method or in the configuration file reader. (Probably the former, so the code here would actually be right.)
- The whole *auto_enable* stuff really belongs to the user-interface part, which is also hard-coded in the auth classes, and should not be.

---

add_params(***params*)

Add the specified keyword arguments as initialization parameters to all the configured auth classes.

Parameters that have already been specified are silently overwritten.

get(*auth_name*, ***kwargs*)

Return an instance of the *Auth* class corresponding to the given *auth_name*, or raise an exception if instanciating the same class has given an unrecoverable exception in past calls.

Additional keyword arguments are passed unchanged to the class constructor and can override values specified at configuration time.

Instances are remembered for the lifetime of the program; if an instance of the given class is already present in the cache, that one is returned; otherwise, an instance is contructed with the given parameters.

> **Caution:** The *params* keyword arguments are only used if a new instance is constructed and are silently ignored if the cached instance is returned.

class gc3libs.authentication.**NoneAuth**(**auth*)
Auth proxy to use when no auth is needed.

gc3libs.authentication.grid

gc3libs.authentication.ssh

Authentication support for accessing resources through the SSH protocol.

gc3libs.backends

Interface to different resource management systems for the GC3Libs.

class gc3libs.backends.**LRMS**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth=None*, ***extra_args*)
Base class for interfacing with a computing resource.

The following construction parameters are also set as instance attributes. All of them are mandatory, except *auth*.

Attribute name	Expected Type	Meaning
name	string	A unique identifier for this resource, used for generating error message.
architecture	set of *Run.Arch* values	Should contain one entry per each architecture supported. Valid architecture values are constants in the *gc3libs.Run.Arch* class.
auth	string	A *gc3libs.authentication.Auth* instance that will be used to access the computational resource associated with this backend. The default value *None* is used to mean that no authentication credentials are needed (e.g., access to the resource has been pre-authenticated) or is managed outside of GC3Pie).
max_cores	int	Maximum number of CPU cores that GC3Pie can allocate on this resource.
max_cores_per_job	int	Maximum number of CPU cores that GC3Pie can allocate on this resource *for a single job*.
max_memory_per_core	Memory	Maximum memory that GC3Pie can allocate to jobs on this resource. The value is *per core*, so the actual amount allocated to a single job is the value of this entry multiplied by the number of cores requested by the job.
max_walltime	Duration	Maximum wall-clock time that can be allotted to a single job running on this resource.

The above should be considered *immutable* attributes: they are specified at construction time and changed never after.

The following attributes are instead dynamically provided (i.e., defined by the *get_resource_status()* method or similar), thus can change over the lifetime of the object:

Attribute name	Type
free_slots	int
user_run	int
user_queued	int
queued	int

static authenticated(*fn*)
Decorator: mark a function as requiring authentication.

Each invocation of the decorated function causes a call to the *get* method of the authentication object (configured with the *auth* parameter to the class constructor).

cancel_job(*app*)
Cancel a running job. If *app* is associated to a queued or running remote job, tell the execution middleware to cancel it.

close()
Implement gracefully close on LRMS dependent resources e.g. transport

free(*app*)
Free up any remote resources used for the execution of *app*. In particular, this should delete any remote directories and files.

Call this method when *app.execution.state* is anything other than *TERMINATED* results in undefined behavior and will likely be the cause of errors later on. Be cautious.

get_resource_status()
Update the status of the resource associated with this *LRMS* instance in-place. Return updated *Resource* object.

get_results(*job*, *download_dir*, *overwrite=False*, *changed_only=True*)
Retrieve job output files into local directory *download_dir*.

Directory *download_dir* must already exists.

If optional 3rd argument *overwrite* is `False` (default), then existing files within *download_dir* (or subdirectories thereof) will *not* be altered in any way.

If *overwrite* is instead `True`, then the (optional) 4th argument *changed_only* determines what files are overwritten:

- if *changed_only* is `True` (default), then only files for which the source has a different size or has been modified more recently than the destination are copied;
- if *changed_only* is `False`, then *all* files in *source* will be copied into *destination*, unconditionally.

Output files that do not exist in *download_dir* will be copied, independently of the *overwrite* and *changed_only* settings.

Parameters

- **job** (Task) – the `Task` instance whose output should be retrieved
- **download_dir** (*str*) – path to download files into
- **overwrite** (*bool*) – if *False*, do not download files that already exist
- **changed_only** (*bool*) – if both this and *overwrite* are *True*, only overwrite those files such that the source is newer or different in size than the destination.

peek(*app*, *remote_filename*, *local_file*, *offset=0*, *size=None*)
Download *size* bytes (at offset *offset* from the start) from remote file *remote_filename* and write them into *local_file*. If *size* is *None* (default), then snarf contents of remote file from *offset* unto the end.

First argument *remote_filename* is the path to a file relative to the remote job "sandbox".

Argument *local_file* is either a local path name (string), or a file-like object supporting a *.write()* method. If *local_file* is a path name, it is created if not existent, otherwise overwritten. In any case, upon exit from this procedure, the stream will be positioned just after the written bytes.

Fourth optional argument *offset* is the offset from the start of the file. If *offset* is negative, it is interpreted as an offset from the *end* of the remote file.

Any exception raised by operations will be re-raised to the caller.

submit_job(*application*, *job*)
Submit an *Application* instance to the configured computational resource; return a *gc3libs.Job* instance for controlling the submitted job.

This method only returns if the job is successfully submitted; upon any failure, an exception is raised.

Note:

1. *job.state* is *not* altered; it is the caller's responsibility to update it.
2. the *job* object may be updated with any information that is necessary for this LRMS to perform further operations on it.

update_job_state(*app*)
Query the state of the remote job associated with *app* and update *app.execution.state* accordingly. Return the corresponding *Run.State*; see *Run.State* for more details.

validate_data(*data_file_list=None*)
Return True if the list of files is expressed in one of the file transfer protocols the LRMS supports.

Return False otherwise.

gc3libs.backends.arc0

gc3libs.backends.arc1

gc3libs.backends.batch

This module provides a generic BatchSystem class from which all batch-like backends should inherit.

class gc3libs.backends.batch.**BatchSystem**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth*, *frontend*, *transport*, *accounting_delay=15*, *ssh_config=None*, *keyfile=None*, *ignore_ssh_host_keys=False*, *ssh_timeout=None*, ***extra_args*)
Base class for backends dealing with a batch-queue system (e.g., PBS/TORQUE, Grid Engine, etc.)

This is an abstract class, that you should subclass in order to interface with a given batch queuing system. (Remember to call this class' constructor in the derived class __init__ method.)

cancel_job(**args*, ***kwargs*)
Cancel a running job. If *app* is associated to a queued or running remote job, tell the execution middleware to cancel it.

close(**args*, ***kwargs*)
Return True if the list of files is expressed in one of the file transfer protocols the LRMS supports.

Return False otherwise.

free(**args*, ***kwargs*)
Free up any remote resources used for the execution of *app*. In particular, this should delete any remote directories and files.

Call this method when *app.execution.state* is anything other than *TERMINATED* results in undefined behavior and will likely be the cause of errors later on. Be cautious.

get_epilogue_script(*app*)
This method will get the epilogue script(s) for the *app* application and will return a string which contains the contents of the script(s) merged together.

get_jobid_from_submit_output(*output*, *regexp*)
Parse the output of the submission command. Regexp is provided by the caller.

get_prologue_script(*app*)
This method will get the prologue script(s) for the *app* application and will return a string which contains the contents of the script(s) merged together.

get_results(**args*, ***kwargs*)
Retrieve job output files into local directory *download_dir*.

Directory *download_dir* must already exists.

If optional 3rd argument *overwrite* is `False` (default), then existing files within *download_dir* (or subdirectories thereof) will *not* be altered in any way.

If *overwrite* is instead `True`, then the (optional) 4th argument *changed_only* determines what files are overwritten:

- if *changed_only* is `True` (default), then only files for which the source has a different size or has been modified more recently than the destination are copied;
- if *changed_only* is `False`, then *all* files in *source* will be copied into *destination*, unconditionally.

Output files that do not exist in *download_dir* will be copied, independently of the *overwrite* and *changed_only* settings.

Parameters

- **job** (Task) – the `Task` instance whose output should be retrieved
- **download_dir** (*str*) – path to download files into
- **overwrite** (*bool*) – if *False*, do not download files that already exist
- **changed_only** (*bool*) – if both this and *overwrite* are *True*, only overwrite those files such that the source is newer or different in size than the destination.

peek(**args*, ***kwargs*)
Download *size* bytes (at offset *offset* from the start) from remote file *remote_filename* and write them into *local_file*. If *size* is *None* (default), then snarf contents of remote file from *offset* unto the end.

First argument *remote_filename* is the path to a file relative to the remote job "sandbox".

Argument *local_file* is either a local path name (string), or a file-like object supporting a *.write()* method. If *local_file* is a path name, it is created if not existent, otherwise overwritten. In any case, upon exit from this procedure, the stream will be positioned just after the written bytes.

Fourth optional argument *offset* is the offset from the start of the file. If *offset* is negative, it is interpreted as an offset from the *end* of the remote file.

Any exception raised by operations will be re-raised to the caller.

submit_job(**args*, ***kwargs*)
This method will create a remote directory to store job's sandbox, and will copy the sandbox in there.

update_job_state(**args*, ***kwargs*)
Query the state of the remote job associated with *app* and update *app.execution.state* accordingly. Return the corresponding *Run.State*; see *Run.State* for more details.

validate_data(*data_file_list*)
Return True if the list of files is expressed in one of the file transfer protocols the LRMS supports.

Return False otherwise.

gc3libs.backends.batch.**generic_filename_mapping**(*jobname*, *jobid*, *file_name*)
Map STDOUT/STDERR filenames (as recorded in *Application.outputs*) to commonly used default STDOUT/STDERR file names (e.g., `<jobname>.o<jobid>`).

gc3libs.backends.lsf

Job control on SGE clusters (possibly connecting to the front-end via SSH).

class gc3libs.backends.lsf.**LsfLrms**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth*, *frontend*, *transport*, *lsf_continuation_line_prefix_length=None*, ***extra_args*)

Job control on LSF clusters (possibly by connecting via SSH to a submit node).

get_resource_status(*obj*, **args*)

Get dynamic information out of the LSF subsystem.

return self

dynamic information required (at least those): total_queued free_slots user_running user_queued

gc3libs.backends.pbs

Job control on PBS/Torque clusters (possibly connecting to the front-end via SSH).

class gc3libs.backends.pbs.**PbsLrms**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth*, *frontend*, *transport*, *queue=None*, ***extra_args*)

Job control on PBS/Torque clusters (possibly by connecting via SSH to a submit node).

get_resource_status(**args*, ***kwargs*)

Update the status of the resource associated with this *LRMS* instance in-place. Return updated *Resource* object.

gc3libs.backends.pbs.**count_jobs**(*qstat_output*, *whoami*)

Parse PBS/Torque's `qstat` output (as contained in string *qstat_output*) and return a quadruple *(R, Q, r, q)* where:

- *R* is the total number of running jobs in the PBS/Torque cell (from any user);
- *Q* is the total number of queued jobs in the PBS/Torque cell (from any user);
- *r* is the number of running jobs submitted by user *whoami*;
- *q* is the number of queued jobs submitted by user *whoami*

gc3libs.backends.sge

Job control on SGE clusters (possibly connecting to the front-end via SSH).

class gc3libs.backends.sge.**SgeLrms**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth*, *frontend*, *transport*, *default_pe=None*, ***extra_args*)

Job control on SGE clusters (possibly by connecting via SSH to a submit node).

get_resource_status(**args*, ***kwargs*)

Update the status of the resource associated with this *LRMS* instance in-place. Return updated *Resource* object.

gc3libs.backends.sge.**compute_nr_of_slots**(*qstat_output*)

Compute the number of total, free, and used/reserved slots from the output of SGE's `qstat -F`.

Return a dictionary instance, mapping each host name into a dictionary instance, mapping the strings `total`, `available`, and `unavailable` to (respectively) the the total number of slots on the host, the number of free slots on the host, and the number of used+reserved slots on the host.

Cluster-wide totals are associated with key `global`.

Note: The 'available slots' computation carried out by this function is unreliable: there is indeed no notion of a 'global' or even 'per-host' number of 'free' slots in SGE. Slot numbers can be computed per-queue, but a host can belong in different queues at the same time; therefore the number of 'free' slots available to a job actually depends on the queue it is submitted to. Since SGE does not force users to submit explicitly to

a queue, rather encourages use of a sort of 'implicit' routing queue, there is no way to compute the number of free slots, as this entirely depends on how local policies will map a job to the available queues.

`gc3libs.backends.sge.`**`count_jobs`**(*qstat_output*, *whoami*)
Parse SGE's `qstat` output (as contained in string *qstat_output*) and return a quadruple *(R, Q, r, q)* where:

- *R* is the total number of running jobs in the SGE cell (from any user);
- *Q* is the total number of queued jobs in the SGE cell (from any user);
- *r* is the number of running jobs submitted by user *whoami*;
- *q* is the number of queued jobs submitted by user *whoami*

`gc3libs.backends.sge.`**`parse_qhost_f`**(*qhost_output*)
Parse SGE's `qhost -F` output (as contained in string *qhost_output*) and return a *dict* instance, mapping each host name to its attributes.

`gc3libs.backends.sge.`**`parse_qstat_f`**(*qstat_output*)
Parse SGE's `qstat -F` output (as contained in string *qstat_output*) and return a *dict* instance, mapping each queue name to its attributes.

gc3libs.backends.shellcmd

Run applications as local processes.

class `gc3libs.backends.shellcmd.`**`ShellcmdLrms`**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth=None*, *frontend='localhost'*, *transport='local'*, *time_cmd=None*, *override='False'*, *spooldir=None*, *resourcedir=None*, *ssh_config=None*, *keyfile=None*, *ignore_ssh_host_keys=False*, *ssh_timeout=None*, ***extra_args*)
Execute an `Application` instance as a local process.

Construction of an instance of *ShellcmdLrms* takes the following optional parameters (in addition to any parameters taken by the base class `LRMS`):

Parameters

- **`time_cmd`** (*str*) – Path to the GNU `time` command. Default is `/usr/bin/time` which is correct on all known Linux distributions.

  This backend uses many of the extended features of GNU `time`, so the shell-builtins or the BSD `time` will not work.
- **`spooldir`** (*str*) – Path to a filesystem location where to create temporary working directories for processes executed through this backend. The default value *None* means to use `$TMPDIR` or `/var/tmp` (see *tempfile.mkftemp* for details).
- **`resourcedir`** (*str*) – Path to a filesystem location where to create a temporary directory that will contain information on the jobs running on the machine. The default value *None* means to use `$HOME/.gc3/shellcmd.d`.
- **`transport`** (*str*) – Transport to use to connecet to the resource. Valid values are *ssh* or *local*.
- **`frontend`** (*str*) – If *transport* is *ssh*, then *frontend* is the hostname of the remote machine where the jobs will be executed.
- **`ignore_ssh_host_key`** (*bool*) – When connecting to a remote resource using *ssh* the server ssh public key is usually checked against a database of known hosts, and if the key is found but it does not match with the one saved in the database the connection will fail. Setting *ignore_ssh_host_key* to *True* will disable this check, thus introducing

a potential security issue, but allowing connection even though the database contain old/invalid keys (the use case is when connecting to VM on a cloud, since the IP is usually reused and therefore the ssh key is recreated).

- **override** (*bool*) – *ShellcmdLrms* by default will try to gather information on the machine the resource is running on, including the number of cores and the available memory. These values may be different from the values stored in the configuration file. If *override* is True, then the values automatically discovered will be used instead of the ones in the configuration file. If *override* is False, instead, the values in the configuration file will be used.
- **ssh_timeout** (*int*) – If *transport* is *ssh*, this value will be used as timeout (in seconds) for the TCP connect.

cancel_job(*app*)
Cancel a running job. If *app* is associated to a queued or running remote job, tell the execution middleware to cancel it.

close()
Implement gracefully close on LRMS dependent resources e.g. transport

free(*app*)
Delete the temporary directory where a child process has run. The temporary directory is removed with all its content, recursively.

If the deletion is successful, the *lrms_execdir* attribute in *app.execution* is reset to *None*; subsequent invocations of this method on the same applications do nothing.

free_slots
Returns the number of cores free

get_resource_status()
Update the status of the resource associated with this *LRMS* instance in-place. Return updated *Resource* object.

get_results(*app*, *download_dir*, *overwrite=False*, *changed_only=True*)
Retrieve job output files into local directory *download_dir*.

Directory *download_dir* must already exists.

If optional 3rd argument *overwrite* is `False` (default), then existing files within *download_dir* (or subdirectories thereof) will *not* be altered in any way.

If *overwrite* is instead `True`, then the (optional) 4th argument *changed_only* determines what files are overwritten:

•if *changed_only* is `True` (default), then only files for which the source has a different size or has been modified more recently than the destination are copied;

•if *changed_only* is `False`, then *all* files in *source* will be copied into *destination*, unconditionally.

Output files that do not exist in *download_dir* will be copied, independently of the *overwrite* and *changed_only* settings.

Parameters

- **job** (Task) – the `Task` instance whose output should be retrieved
- **download_dir** (*str*) – path to download files into
- **overwrite** (*bool*) – if *False*, do not download files that already exist
- **changed_only** (*bool*) – if both this and *overwrite* are *True*, only overwrite those files such that the source is newer or different in size than the destination.

peek(*app*, *remote_filename*, *local_file*, *offset=0*, *size=None*)
Download *size* bytes (at offset *offset* from the start) from remote file *remote_filename* and write them into *local_file*. If *size* is *None* (default), then snarf contents of remote file from *offset* unto the end.

First argument *remote_filename* is the path to a file relative to the remote job "sandbox".

Argument *local_file* is either a local path name (string), or a file-like object supporting a *.write()* method. If *local_file* is a path name, it is created if not existent, otherwise overwritten. In any case, upon exit from this procedure, the stream will be positioned just after the written bytes.

Fourth optional argument *offset* is the offset from the start of the file. If *offset* is negative, it is interpreted as an offset from the *end* of the remote file.

Any exception raised by operations will be re-raised to the caller.

submit_job(*app*)
Run an *Application* instance as a local process.

See *LRMS.submit_job*

update_job_state(*app*)
Query the running status of the local process whose PID is stored into *app.execution.lrms_jobid*, and map the POSIX process status to GC3Libs *Run.State*.

validate_data(*data_file_list=[]*)
Return *False* if any of the URLs in *data_file_list* cannot be handled by this backend.

The `shellcmd` backend can only handle `file` URLs.

gc3libs.backends.slurm

Job control on SLURM clusters (possibly connecting to the front-end via SSH).

class gc3libs.backends.slurm.**SlurmLrms**(*name*, *architecture*, *max_cores*, *max_cores_per_job*, *max_memory_per_core*, *max_walltime*, *auth*, *frontend*, *transport*, ***extra_args*)
Job control on SLURM clusters (possibly by connecting via SSH to a submit node).

get_resource_status(**args*, ***kwargs*)
Update the status of the resource associated with this *LRMS* instance in-place. Return updated *Resource* object.

gc3libs.backends.slurm.**count_jobs**(*squeue_output*, *whoami*)
Parse SLURM's `squeue` output and return a quadruple *(R, Q, r, q)* where:

- *R* is the total number of running jobs (from any user);
- *Q* is the total number of queued jobs (from any user);
- *r* is the number of running jobs submitted by user *whoami*;
- *q* is the number of queued jobs submitted by user *whoami*

The *squeue_output* must contain the results of an invocation of `squeue --noheader --format='%i^%T^%u^%U^%r^%R'`.

gc3libs.backends.transport

The *Transport* class hierarchy provides an abstraction layer to execute commands and copy/move files irrespective of whether the destination is the local computer or a remote front-end that we access via SSH.

gc3libs.cmdline

Prototype classes for GC3Libs-based scripts.

Classes implemented in this file provide common and recurring functionality for GC3Libs command-line utilities and scripts. User applications should implement their specific behavior by subclassing and overriding a few customization methods.

There are currently two public classes provided here:

GC3UtilsScript Base class for all the GC3Utils commands. Implements a few methods useful for writing command-line scripts that operate on jobs by ID.

SessionBasedScript Base class for the `grosetta`/`ggamess`/`gcodeml` scripts. Implements a long-running script to submit and manage a large number of jobs grouped into a "session".

class `gc3libs.cmdline.`**`GC3UtilsScript`**`(`***extra_args*`)`
Base class for GC3Utils scripts.

The default command line implemented is the following:

> script [options] JOBID [JOBID ...]

By default, only the standard options `-h`/`--help` and `-V`/`--version` are considered; to add more, override `setup_options()` To change default positional argument parsing, override *`setup_args()`*

`pre_run``()`
Perform parsing of standard command-line options and call into *parse_args()* to do non-optional argument processing.

`setup``()`
Setup standard command-line parsing.

GC3Utils scripts should probably override *`setup_args()`* and `setup_options()` to modify command-line parsing.

`setup_args``()`
Set up command-line argument parsing.

The default command line parsing considers every argument as a job ID; actual processing of the IDs is done in `parse_args()`

class `gc3libs.cmdline.`**`SessionBasedScript`**`(`***extra_args*`)`
Base class for `grosetta`/`ggamess`/`gcodeml` and like scripts. Implements a long-running script to submit and manage a large number of jobs grouped into a "session".

The generic scripts implements a command-line like the following:

```
PROG [options] INPUT [INPUT ...]
```

First, the script builds a list of input files by recursively scanning each of the given INPUT arguments for files matching the *self.input_file_pattern* glob string (you can set it via a keyword argument to the ctor). To perform a different treatment of the command-line arguments, override the *`process_args()`* method.

Then, new jobs are added to the session, based on the results of the *process_args()* method above. For each tuple of items returned by *process_args()*, an instance of class *self.application* (which you can set by a keyword argument to the ctor) is created, passing it the tuple as init args, and added to the session.

The script finally proceeds to updating the status of all jobs in the session, submitting new ones and retrieving output as needed. When all jobs are done, the method `done()` is called, and its return value is used as the script's exit code.

The script's exitcode tracks job status, in the following way. The exitcode is a bitfield; only the 4 least-significant bits are used, with the following meaning:

Bit	Meaning
0	Set if a fatal error occurred: the script could not complete
1	Set if there are jobs in *FAILED* state
2	Set if there are jobs in *RUNNING* or *SUBMITTED* state
3	Set if there are jobs in *NEW* state

This boils down to the following rules:

- exitcode == 0: all jobs terminated successfully, no further action
- exitcode == 1: an error interrupted script execution

- exitcode == 2: all jobs terminated, not all of them successfully
- exitcode > 3: run the script again to progress jobs

after_main_loop()
Hook executed after exit from the main loop.

This is called after the main loop has exited (for whatever reason), but *before* the session is finally saved and other connections are finalized.

Override in subclasses to plug any behavior here; the default implementation does nothing.

before_main_loop()
Hook executed before entering the scripts' main loop.

This is the last chance to alter the script state as it will be seen by the main loop.

Override in subclasses to plug any behavior here; the default implementation does nothing.

every_main_loop()
Hook executed during each round of the main loop.

This is called from within the main loop, after progressing all tasks.

Override in subclasses to plug any behavior here; the default implementation does nothing.

input_filename_pattern = None
IN *SessionBasedScript* CONSTRUCTOR

make_directory_path(*pathspec*, *jobname*)
Return a path to a directory, suitable for storing the output of a job (named after *jobname*). It is not required that the returned path points to an existing directory.

This is called by the default *process_args()* using *self.params.output* (i.e., the argument to the `-o`/`--output` option) as *pathspec*, and *jobname* and *args* exactly as returned by *new_tasks()*

The default implementation substitutes the following strings within *pathspec*:

- `SESSION` is replaced with the name of the current session (as specified by the `-s`/`--session` command-line option) with a suffix `.out` appended;
- `NAME` is replaced with *jobname*;
- `DATE` is replaced with the current date, in *YYYY-MM-DD* format;
- `TIME` is replaced with the current time, in *HH:MM* format.

make_task_controller()
Return a 'Controller' object to be used for progressing tasks and getting statistics. In detail, a good 'Controller' object has to implement *progress* and *stats* methods with the same interface as *gc3libs.core.Engine*.

By the time this method is called (from `_main()`), the following instance attributes are already defined:

- *self._core*: a *gc3libs.core.Core* instance;
- *self.session*: the *gc3libs.session.Session* instance that should be used to save/load jobs

In addition, any other attribute created during initialization and command-line parsing is of course available.

new_tasks(*extra*)
Iterate over jobs that should be added to the current session. Each item yielded must have the form *(jobname, cls, args, kwargs)*, where:

- *jobname* is a string uniquely identifying the job in the session; if a job with the same name already exists, this item will be ignored.
- *cls* is a callable that returns an instance of *gc3libs.Application* when called as *cls(*args, **kwargs)*.

- *args* is a tuple of arguments for calling *cls*.
- *kwargs* is a dictionary used to provide keyword arguments when calling *cls*.

This method is called by the default `process_args()`, passing *self.extra* as the *extra* parameter.

The default implementation of this method scans the arguments on the command-line for files matching the glob pattern *self.input_filename_pattern*, and for each matching file returns a job name formed by the base name of the file (sans extension), the class given by *self.application*, and the full path to the input file as sole argument.

If *self.instances_per_file* and *self.instances_per_job* are set to a value other than 1, for each matching file N jobs are generated, where N is the quotient of *self.instances_per_file* by *self.instances_per_job*.

See also: `process_args()`

pre_run()
Perform parsing of standard command-line options and call into *parse_args()* to do non-optional argument processing.

print_summary_table(*output*, *stats*)
Print a text summary of the session status to *output*. This is used to provide the "normal" output of the script; when the `-l` option is given, the output of the *print_tasks_table* function is appended.

Override this in subclasses to customize the report that you provide to users. By default, this prints a table with the count of tasks for each possible state.

The *output* argument is a file-like object, only the *write* method of which is used. The *stats* argument is a dictionary, mapping each possible *Run.State* to the count of tasks in that state; see *Engine.stats* for a detailed description.

print_tasks_table(*output=<open file '<stdout>', mode 'w'>*, *states=Enum(['TERMINATED', 'UNKNOWN', 'SUBMITTED', 'RUNNING', 'TERMINATING', 'STOPPED', 'NEW'])*, *only=<type 'object'>*)
Output a text table to stream *output*, giving details about tasks in the given states.

Optional second argument *states* restricts the listing to tasks that are in one of the specified states. By default, all task states are allowed. The *states* argument should be a list or a set of *Run.State* values.

Optional third argument *only* further restricts the listing to tasks that are instances of a subclass of *only*. By default, there is no restriction and all tasks are listed. The *only* argument can be a Python class or a tuple – anything infact, that you can pass as second argument to the *isinstance* operator.

Parameters

- **output** – An output stream (file-like object)
- **states** – List of states (*Run.State* items) to consider.
- **only** – Root class (or tuple of root classes) of tasks to consider.

process_args()
Process command-line positional arguments and set up the session accordingly. In particular, new jobs should be added to the session during the execution of this method: additions are not contemplated elsewhere.

This method is called by the standard `_main()` after loading or creating a session into *self.session*. New jobs should be appended to *self.session* and it is also permitted to remove existing ones.

The default implementation calls `new_tasks()` and adds to the session all jobs whose name does not clash with the jobname of an already existing task.

See also: `new_tasks()`

setup()
Setup standard command-line parsing.

GC3Libs scripts should probably override `setup_args()` to modify command-line parsing.

setup_args()
Set up command-line argument parsing.

The default command line parsing considers every argument as an (input) path name; processing of the given path names is done in `parse_args()`

gc3libs.cmdline.**nonnegative_int**(*num*)
This function raise an ArgumentTypeError if *num* is a negative integer (<0), and returns int(num) otherwise. *num* can be any object which can be converted to an int.

```
>>> nonnegative_int('1')
1
>>> nonnegative_int(1)
1
>>> nonnegative_int('-1')
Traceback (most recent call last):
    ...
ArgumentTypeError: '-1' is not a non-negative integer number.
>>> nonnegative_int(-1)
Traceback (most recent call last):
    ...
ArgumentTypeError: '-1' is not a non-negative integer number.
```

Please note that *0* and *'-0'* are ok:

```
>>> nonnegative_int(0)
0
>>> nonnegative_int(-0)
0
>>> nonnegative_int('0')
0
>>> nonnegative_int('-0')
0
```

Floats are ok too:

```
>>> nonnegative_int(3.14)
3
>>> nonnegative_int(0.1)
0
```

```
>>> nonnegative_int('ThisWillRaiseAnException')
Traceback (most recent call last):
    ...
ArgumentTypeError: 'ThisWillRaiseAnException' is not a non-negative ...
```

gc3libs.cmdline.**positive_int**(*num*)
This function raises an ArgumentTypeError if *num* is not a*strictly* positive integer (>0) and returns int(num) otherwise. *num* can be any object which can be converted to an int.

```
>>> positive_int('1')
1
>>> positive_int(1)
1
>>> positive_int('-1')
Traceback (most recent call last):
...
ArgumentTypeError: '-1' is not a positive integer number.
>>> positive_int(-1)
Traceback (most recent call last):
...
ArgumentTypeError: '-1' is not a positive integer number.
>>> positive_int(0)
Traceback (most recent call last):
```

```
...
ArgumentTypeError: '0' is not a positive integer number.
```

Floats are ok too:

```
>>> positive_int(3.14)
3
```

but please take care that float *greater* than 0 will fail:

```
>>> positive_int(0.1)
Traceback (most recent call last):
...
ArgumentTypeError: '0.1' is not a positive integer number.
```

Please note that *0* is NOT ok:

```
>>> positive_int(-0)
Traceback (most recent call last):
...
ArgumentTypeError: '0' is not a positive integer number.
>>> positive_int('0')
Traceback (most recent call last):
...
ArgumentTypeError: '0' is not a positive integer number.
>>> positive_int('-0')
Traceback (most recent call last):
...
ArgumentTypeError: '-0' is not a positive integer number.
```

Any string which does cannot be converted to an integer will fail:

```
>>> positive_int('ThisWillRaiseAnException')
Traceback (most recent call last):
    ...
ArgumentTypeError: 'ThisWillRaiseAnException' is not a positive integer ...
```

gc3libs.config

Deal with GC3Pie configuration files.

class gc3libs.config.**Configuration**(**locations*, ***extra_args*)

In-memory representation of the GC3Pie configuration.

This class provides facilities for:

- parsing configuration files (methods *load()* and *merge_file()*);
- validating the loaded values;
- instanciating the internal GC3Pie objects resulting from the configuration (methods *make_auth()* and make_resource()).

The constructor takes a list of files to load (*locations*) and a list of key=value pairs to provide defaults for the configuration. Both lists are optional and can be omitted, resulting in a configuration containing only GC3Pie default values.

Example 1: initialization from config file:

```
>>> import os
>>> example_cfgfile = os.path.join(
...     os.path.dirname(__file__), 'etc/gc3pie.conf.example')
>>> cfg = Configuration(example_cfgfile)
>>> cfg.debug
'0'
```

Example 2: initialization from key=value list:

```
>>> cfg = Configuration(auto_enable_auth=False, foo=1, bar='baz')
>>> cfg.auto_enable_auth
False
>>> cfg.foo
1
>>> cfg.bar
'baz'
```

When both a configuration file *and* a key=value list is present, values in the configuration files override those in the key=value list:

```
>>> cfg = Configuration(example_cfgfile, debug=1)
>>> cfg.debug
'0'
```

Example 3: default initialization:

```
>>> cfg = Configuration()
>>> cfg.auto_enable_auth
True
```

auth_factory
The instance of *gc3libs.authentication.Auth* used to manage auth access for the resources.

This is a *read-only* attribute, created upon first access with the values set in *self.auths* and *self.auto_enabled*.

load(**locations*)
Merge settings from configuration files into this *Configuration* instance.

Environment variables and ~ references are expanded in the location file names.

If any of the specified files does not exist or cannot be read (for whatever reason), a message is logged but the error is ignored. However, a *NoConfigurationFile* exception is raised if *none* of the specified locations could be read.

Raises *gc3libs.exceptions.NoConfigurationFile* if none of the specified files could be read.

make_auth(*name*)
Return factory for auth credentials configured in section `[auth/name]`.

make_resources(*ignore_errors=True*)
Make backend objects corresponding to the configured resources.

Return a dictionary, mapping the resource name (string) into the corresponding backend object.

By default, errors in constructing backends (e.g., due to a bad configuration) are silently ignored: the offending configuration is just dropped. This can be changed by setting the optional argument *ignore_errors* to *False*: in this case, an exception is raised whenever we fail to construct a backend.

merge_file(*filename*)
Read configuration files and merge the settings into this *Configuration* object.

Contrary to *load()* (which see), the file name is taken literally and an error is raised if the file cannot be read for whatever reason.

Any parameter which is set in the configuration files `[DEFAULT]` section, and whose name does not start with underscore (_) defines an attribute in the current *Configuration*.

Warning: No type conversion is performed on values set this way - so they all end up being strings!

Raises *gc3libs.exceptions.ConfigurationError* if the configuration file does not exist, cannot be read, is corrupt or has wrong format.

gc3libs.core

Top-level interface to Grid functionality.

class gc3libs.core.**Core**(*cfg*, *matchmaker=<gc3libs.core.MatchMaker object>*, *resource_errors_are_fatal=None*)

Core operations: submit, update state, retrieve (a snapshot of) output, cancel job.

Core operations are *blocking*, i.e., they return only after the operation has successfully completed, or an error has been detected.

Operations are always performed by a *Core* object. *Core* implements an overlay Grid on the resources specified in the configuration file.

Initialization of a *Core* instance also initializes all resources in the passed `Configuration` instance. By default, GC3Pie's *Core* objects will ignore errors in initializing resources, and only raise an exception if *no* resources can be initialized. This can be changed by either passing an optional argument `resource_errors_are_fatal=True`, or by setting the environmental variable `GC3PIE_RESOURCE_INIT_ERRORS_ARE_FATAL` to `yes` or `1`.

add(*task*)

This method is here just to allow *Core* and *Engine* objects to be used interchangeably. It's effectively a no-op, as it makes no sense in the synchronous/blocking semantics implemented by *Core*.

close()

Used to invoke explicitly the destructor on objects e.g. LRMS

fetch_output(*app*, *download_dir=None*, *overwrite=False*, *changed_only=True*, ***extra_args*)

Retrieve output into local directory *app.output_dir*.

If the task is not expected to produce any output (i.e., *app.would_output == False*) then the only effect of this is to advance the state of `TERMINATING` tasks to `TERMINATED`.

Optional argument *download_dir* overrides the download location.

The download directory is created if it does not exist. If it already exists, and the optional argument *overwrite* is `False` (default), it is renamed with a *.NUMBER* suffix and a new empty one is created in its place. Otherwise, if 'overwrite' is `True`, files are downloaded over the ones already present; in this case, the *changed_only* argument controls which files are overwritten:

- if *changed_only* is `True` (default), then only files for which the source has a different size or has been modified more recently than the destination are copied;
- if *changed_only* is `False`, then *all* files in *source* will be copied into *destination*, unconditionally.

Source files that do not exist at *destination* will be copied, independently of the *overwrite* and *changed_only* settings.

If the task is in TERMINATING state, the state is changed to *TERMINATED*, attribute `output_dir` is set to the absolute path to the directory where files were downloaded, and the *terminated* transition method is called on the *app* object.

Task output cannot be retrieved when *app.execution* is in one of the states *NEW* or *SUBMITTED*; an *OutputNotAvailableError* exception is thrown in these cases.

Raise *gc3libs.exceptions.OutputNotAvailableError* if no output can be fetched from the remote job (e.g., the Application/Task object is in *NEW* or *SUBMITTED* state, indicating the remote job has not started running).

free(*app*, ***extra_args*)

Free up any remote resources used for the execution of *app*. In particular, this should delete any remote directories and files.

It is an error to call this method if *app.execution.state* is anything other than *TERMINATED*: an *InvalidOperation* exception will be raised in this case.

Raise *gc3libs.exceptions.InvalidOperation* if *app.execution.state* differs from *Run.State.TERMINATED*.

get_resources(***extra_args*)
Return list of resources configured into this *Core* instance.

kill(*app*, ***extra_args*)
Terminate a job.

Terminating a job in RUNNING, SUBMITTED, or STOPPED state entails canceling the job with the remote execution system; terminating a job in the NEW or TERMINATED state is a no-op.

peek(*app*, *what='stdout'*, *offset=0*, *size=None*, ***extra_args*)
Download *size* bytes (at *offset* bytes from the start) from the remote job standard output or error stream, and write them into a local file. Return file-like object from which the downloaded contents can be read.

If *size* is *None* (default), then snarf all available contents of the remote stream from *offset* unto the end.

The only allowed values for the *what* arguments are the strings *'stdout'* and *'stderr'*, indicating that the relevant section of the job's standard output resp. standard error should be downloaded.

remove(*task*)
This method is here just to allow *Core* and *Engine* objects to be used interchangeably. It's effectively a no-op, as it makes no sense in the synchronous/blocking semantics implemented by *Core*.

select_resource(*match*)
Disable resources that *do not* satisfy predicate *match*. Return number of enabled resources.

Argument *match* can be:

- either a function (or a generic callable) that is passed each *Resource* object in turn, and should return a boolean indicating whether the resources should be kept (*True*) or not (*False*);
- or it can be a string: only resources whose name matches (wildcards `*` and `?` are allowed) are retained.

Note: Calling this method modifies the configured list of resources in-place.

---

submit(*app*, *resubmit=False*, *targets=None*, ***extra_args*)
Submit a job running an instance of the given task *app*.

Upon successful submission, call the *submitted* method on the *app* object. If *targets* are given, submission of the task is attempted to the resources in the order given; the *submit* method returns after the first successful attempt. If *targets* is `None` (default), a brokering procedure is run to determine the best resource among the configured ones.

At the beginning of the submission process, the *app.execution* state is reset to `NEW`; if submission is successful, the task will be in `SUBMITTED` or `RUNNING` state when this call returns.

Raise *gc3libs.exceptions.InputFileError* if an input file does not exist or cannot otherwise be read.

Parameters

- **app** (Task) – A GC3Pie `Task` instance to be submitted.
- **resubmit** – If `True`, submit task regardless of its execution state; if `False` (default), submission is a no-op if task is not in `NEW` state.
- **targets** (list) – A list of *Resource's to submit the task to; resources are tried in the order given. If ''None'* (default), perform brokering among all the configured resources.

update_job_state(**apps*, ***extra_args*)
Update state of all applications passed in as arguments.

If keyword argument *update_on_error* is *False* (default), then application execution state is not changed in case a backend error happens; it is changed to *UNKNOWN* otherwise.

Note that if state of a job changes, the *Run.state* calls the appropriate handler method on the application/task object.

Raise *gc3libs.exceptions.InvalidArgument* in case one of the passed *Application* or *Task* objects is invalid. This can stop updating the state of other objects in the argument list.

Raise *gc3libs.exceptions.ConfigurationError* if the configuration of this *Core* object is invalid or otherwise inconsistent (e.g., a resource references a non-existing auth section).

update_resources(*resources=<built-in function all>*, ***extra_args*)

Update the state of a given set of resources.

Each resource object in the returned list will have its *updated* attribute set to *True* if the update operation succeeded, or *False* if it failed.

Optional argument *resources* should be a subset of the resources configured in this *Core* instance (the actual `Lrms` objects, not the resource names). By default, all configured resources are updated.

class gc3libs.core.**Engine**(*controller*, *tasks=[]*, *store=None*, *can_submit=True*, *can_retrieve=True*, *max_in_flight=0*, *max_submitted=0*, *output_dir=None*, *scheduler=<gc3libs.core.scheduler object>*, *retrieve_running=False*, *retrieve_overwrites=False*, *retrieve_changed_only=True*)

Submit tasks in a collection, and update their state until a terminal state is reached. Specifically:

- tasks in *NEW* state are submitted;
- the state of tasks in *SUBMITTED*, *RUNNING* or *STOPPED* state is updated;
- when a task reaches *TERMINATED* state, its output is downloaded.

The behavior of *Engine* instances can be further customized by setting the following instance attributes:

can_submit Boolean value: if *False*, no task will be submitted.

can_retrieve Boolean value: if *False*, no output will ever be retrieved.

max_in_flight If >0, limit the number of tasks in *SUBMITTED* or *RUNNING* state: if the number of tasks in *SUBMITTED*, *RUNNING* or *STOPPED* state is greater than *max_in_flight*, then no new submissions will be attempted.

max_submitted If >0, limit the number of tasks in *SUBMITTED* state: if the number of tasks in *SUBMITTED*, *RUNNING* or *STOPPED* state is greater than *max_submitted*, then no new submissions will be attempted.

output_dir Base directory for job output; if not *None*, each task's results will be downloaded in a subdirectory named after the task's *permanent_id*.

scheduler A factory function for creating objects that conform to the *Scheduler* interface to control task submission; see the *Scheduler* documentation for details. The default value implements a first-come first-serve algorithm: tasks are submitted in the order they have been added to the *Engine*.

retrieve_running If `True`, snapshot output from RUNNING jobs at every invocation of *progress()*

retrieve_overwrites If `True`, overwrite files in the output directory of any job (as opposed to moving destination away and downloading a fresh copy). See *Core.fetch_output()* for details.

retrieve_changed_only If both this and *overwrite* are `True`, then only changed files are downloaded. See *Core.fetch_output()* for details.

Any of the above can also be set by passing a keyword argument to the constructor (assume `g` is a *Core* instance):

```
| >>> e = Engine(g, can_submit=False)
| >>> e.can_submit
| False
```

add(*task*)
Add *task* to the list of tasks managed by this Engine. Adding a task that has already been added to this *Engine* instance results in a no-op.

close()
Call explicilty finalize methods on relevant objects e.g. LRMS

fetch_output(*task*, *output_dir=None*, *overwrite=False*, *changed_only=True*, ***extra_args*)
Enqueue task for later output retrieval.

> **Warning:** FIXME
> The *output_dir*, *overwrite*, and *changed_only* parameters are currently ignored.

free(*task*, ***extra_args*)
Proxy for *Core.free*, which see.

get_resources()
Return list of resources configured into this *Core* instance.

kill(*task*, ***extra_args*)
Schedule a task for killing on the next *progress* run.

peek(*task*, *what='stdout'*, *offset=0*, *size=None*, ***extra_args*)
Proxy for *Core.peek* (which see).

progress()
Update state of all registered tasks and take appropriate action. Specifically:

- tasks in *NEW* state are submitted;
- the state of tasks in *SUBMITTED*, *RUNNING*, *STOPPED* or *UNKNOWN* state is updated;
- when a task reaches *TERMINATING* state, its output is downloaded.
- tasks in *TERMINATED* status are simply ignored.

The *max_in_flight* and *max_submitted* limits (if >0) are taken into account when attempting submission of tasks.

remove(*task*)
Remove a *task* from the list of tasks managed by this Engine.

select_resource(*match*)
Disable resources that *do not* satisfy predicate *match*. Return number of enabled resources.

Argument *match* can be:

- either a function (or a generic callable) that is passed each *Resource* object in turn, and should return a boolean indicating whether the resources should be kept (*True*) or not (*False*);
- or it can be a string: only resources whose name matches (wildcards `*` and `?` are allowed) are retained.

Note: Calling this method modifies the configured list of resources in-place.

---

stats(*only=None*)
Return a dictionary mapping each state name into the count of tasks in that state. In addition, the following keys are defined:

- *ok*: count of TERMINATED tasks with return code 0
- *failed*: count of TERMINATED tasks with nonzero return code
- *total*: total count of managed tasks, whatever their state

If the optional argument *only* is not None, tasks whose whose class is not contained in *only* are ignored.
: param tuple only: Restrict counting to tasks of these classes.

submit (*task*, *resubmit=False*, *targets=None*, ***extra_args*)
Submit *task* at the next invocation of *progress*.

The *task* state is reset to `NEW` and then added to the collection of managed tasks.

The *targets* argument is only present for interface compatiblity with `Core.submit()` but is otherwise ignored.

update_job_state (**tasks*, ***extra_args*)
Return list of *current* states of the given tasks. States will only be updated at the next invocation of *progress*; in particular, no state-change handlers are called as a result of calling this method.

class gc3libs.core.**MatchMaker**
Select and sort resources for attempting submission of a *Task*.

A match-making algorithm must implement two methods:

- *filter*: given a task and a list of resources, return the list of resources that the given task could be submitted to.
- *rank*: given a task and a list of resources, return a list of resources sorted in preference order, i.e., submission of the given task will be attempted to the first returned resource, then the next one, etc.

This class implements the default match-making algorithm in GC3Pie, which operates as follows:

- *filter phase:* if *task* has a *compatible_resources* method (as instances of `Application` do), retain only those resources where it evaluates to `True`. Otherwise, return the resources list unchanged.
- *rank phase:* sort resources according to the task's *rank_resources* method, or retain the given order if task does not define such method.

filter (*task*, *resources*)
Return the subset of resources to which *task* could be submitted to.

Note that the result subset could be empty (no resource can accomodate task's requirements).

The default implementation uses the task's *compatible_resources* method to retain only the resources that satisfy the task's requirements. If *task* does not provide such a method, the resource list is returned unchanged.

rank (*task*, *resources*)
Sort the list of *resources* in the preferred order for submitting *task*.

Unless overridden in a derived class, this calls the task's *rank_resources* method to sort the list. If the task does not provide such a method, the resources list is returned unchanged.

class gc3libs.core.**Scheduler** (*tasks*, *resources*)
Instances of the *Scheduler* class are used in `Engine.progress()` to determine what tasks (among those in *Run.State.NEW* state) are to be submitted.

A *Scheduler* object must implement *both* the context protocol *and* the iterator protocol.

The way a *Scheduler* instance is actually used within *Engine* is as follows:

0. A *Scheduler* instance is created, passing it two arguments: a list of tasks in `NEW` state, and a dictionary of configured resources (keys are resource names, values are actual resource objects).
1. When a new submission cycle starts, the `__enter__()` method is called.
2. The *Engine* iterates by repeatedly calling the `next()` method to receive tasks to be submitted. The `send()` and `throw()` methods are used to notify the scheduler of the outcome of the submission attempt.
3. When the submission cycle ends, the `__exit__()` method is called.

The *Scheduler.schedule* generator is the heart of the submission process and has basically complete control over it. It is initialized with the list of tasks in `NEW` state, and the list of configured resources. The `next()` method should yield pairs *(task index, resource name)*, where the *task index* is the position of the task to be submitted next in the given list, and –similarly– the *resource name* is the name of the resource to which the task should be submitted.

For each pair yielded, submission of that task to the selected resource is attempted; the state of the task object after submission is sent back (via the `send()` method) to the *Scheduler* instance; if an exception is raised, that exception is thrown (via the `throw()` method) into the scheduler object instead. Submission stops when the *next()* call raises a *StopIteration* exception.

class gc3libs.core.**scheduler**(*fn*)
Decorate a generator function for use as a *Scheduler* object.

gc3libs.debug

Tools for debugging GC3Libs based programs.

Part of the code used in this module originally comes from:

- http://wordaligned.com/articles/echo

gc3libs.debug.**format_arg_value**(*arg_val*)
Return a string representing a (name, value) pair.

Example:

```
>>> format_arg_value(('x', (1, 2, 3)))
'x=(1, 2, 3)'
```

gc3libs.debug.**is_class_private_name**(*name*)
Determine if a name is a class private name.

gc3libs.debug.**is_classmethod**(*instancemethod*)
Determine if an instancemethod is a classmethod.

gc3libs.debug.**method_name**(*method*)
Return a method's name.

This function returns the name the method is accessed by from outside the class (i.e. it prefixes "private" methods appropriately).

gc3libs.debug.**name**(*item*)
Return an item's name.

gc3libs.debug.**trace**(*fn*, *log=<bound method Logger.debug of <logging.Logger object at 0x7f0dc48d9150>>*)
Logs calls to a function.

Returns a decorated version of the input function which "echoes" calls made to it by writing out the function's name and the arguments it was called with.

gc3libs.debug.**trace_class**(*cls*, *log=<bound method Logger.debug of <logging.Logger object at 0x7f0dc48d9150>>*)
Trace calls to class methods and static functions

gc3libs.debug.**trace_instancemethod**(*cls*, *method*, *log=<bound method Logger.debug of <logging.Logger object at 0x7f0dc48d9150>>*)
Change an instancemethod so that calls to it are traced.

Replacing a classmethod is a little more tricky. See: http://www.python.org/doc/current/ref/types.html

gc3libs.debug.**trace_module**(*mod*, *log=<bound method Logger.debug of <logging.Logger object at 0x7f0dc48d9150>>*)
Trace calls to functions and methods in a module.

gc3libs.exceptions

Exceptions specific to the *gc3libs* package.

In addition to the exceptions listed here, `gc3libs` functions try to use Python builtin exceptions with the same meaning they have in core Python, namely:

- *TypeError* is raised when an argument to a function or method has an incompatible type or does not implement the required protocol (e.g., a number is given where a sequence is expected).
- *ValueError'is raised when an argument to a function or method has the correct type, but fails to satisfy other constraints in the function contract (e.g., a positive number is required, and '-1* is passed instead).
- *AssertionError* is raised when some internal assumption regarding state or function/method calling contract is violated. Informally, this indicates a bug in the software.

exception `gc3libs.exceptions.`**`ApplicationDescriptionError`**(*msg*, *do_log=True*)
Raised when the dumped description on a given Application produces something that the LRMS backend cannot process.

exception `gc3libs.exceptions.`**`AuthError`**(*msg*, *do_log=False*)
Base class for Auth-related errors.

Should *never* be instanciated: create a specific error class describing the actual error condition.

exception `gc3libs.exceptions.`**`AuxiliaryCommandError`**(*msg*, *do_log=False*)
Raised when some external command that we depend upon has failed.

For instance, we might need to list processes on a remote machine but `ps aux` does not run because of insufficient privileges.

exception `gc3libs.exceptions.`**`ConfigurationError`**(*msg*, *do_log=True*)
Raised when the configuration file (or parts of it) could not be read/parsed. Also used to signal that a required parameter is missing or has an unknown/invalid value.

exception `gc3libs.exceptions.`**`ConfigurationFileError`**(*msg*, *do_log=True*)
Generic issue with the configuration file(s).

exception `gc3libs.exceptions.`**`CopyError`**(*source*, *destination*, *ex*)
Error copying a file from *source* to 'destination.

exception `gc3libs.exceptions.`**`DataStagingError`**(*msg*, *do_log=False*)
Base class for data staging and movement errors.

Should *never* be instanciated: create a specific error class describing the actual error condition.

exception `gc3libs.exceptions.`**`DetachedFromGridError`**(*msg*, *do_log=False*)
Raised when a method (other than `attach()`) is called on a detached *Task* instance.

exception `gc3libs.exceptions.`**`DuplicateEntryError`**(*msg*, *do_log=False*)
Raised by *Application.__init__* if not all (local or remote) entries in the input or output files are distinct.

exception `gc3libs.exceptions.`**`Error`**(*msg*, *do_log=False*)
Base class for all error-level exceptions in GC3Pie.

Generally, this indicates a non-fatal error: depending on the nature of the task, steps could be taken to continue, but users *must* be aware that an error condition occurred, so the message is sent to the logs at the ERROR level.

Exceptions indicating an error condition after which the program cannot continue and should immediately stop, should use the `FatalError` base class.

exception `gc3libs.exceptions.`**`FatalError`**(*msg*, *do_log=True*)
A fatal error: execution cannot continue and program should report to user and then stop.

The message is sent to the logs at CRITICAL level when the exception is first constructed.

This is the base class for all fatal exceptions.

exception gc3libs.exceptions.**InputFileError**(*msg*, *do_log=True*)
Raised when an input file is specified, which does not exist or cannot be read.

exception gc3libs.exceptions.**InternalError**(*msg*, *do_log=False*)
Raised when some function cannot fulfill its duties, for reasons that do not depend on the library client code. For instance, when a response string gotten from an external command cannot be parsed as expected.

exception gc3libs.exceptions.**InvalidArgument**(*msg*, *do_log=False*)
Raised when the arguments passed to a function do not honor some required contract. For instance, either one of two optional arguments must be provided, but none of them was.

exception gc3libs.exceptions.**InvalidOperation**(*msg*, *do_log=False*)
Raised when an operation is attempted, that is not considered valid according to the system state. For instance, trying to retrieve the output of a job that has not yet been submitted.

exception gc3libs.exceptions.**InvalidResourceName**(*msg*, *do_log=False*)
Raised to signal that no computational resource with the given name is defined in the configuration file.

exception gc3libs.exceptions.**InvalidType**(*msg*, *do_log=False*)
A specialization of‘InvalidArgument‘ for cases when the type of the passed argument does not match expectations.

exception gc3libs.exceptions.**InvalidUsage**(*msg*, *do_log=True*)
Raised when a command is not provided all required arguments on the command line, or the arguments do not match the expected syntax.

Since the exception message is the last thing a user will see, try to be specific about what is wrong on the command line.

exception gc3libs.exceptions.**InvalidValue**(*msg*, *do_log=False*)
A specialization of‘InvalidArgument‘ for cases when the type of the passed argument does not match expectations.

exception gc3libs.exceptions.**LRMSSkipSubmissionToNextIteration**(*msg*, *do_log=False*)
An elastic resource has initiated adapting for a new task. Although we cannot submit the task right now, it *will* be accepted in the (not too distant) future.

exception gc3libs.exceptions.**LoadError**(*msg*, *do_log=False*)
Raised upon errors loading a job from the persistent storage.

exception gc3libs.exceptions.**MaximumCapacityReached**(*msg*, *do_log=False*)
Indicates that a resource is full and cannot run any more jobs.

exception gc3libs.exceptions.**NoAccessibleConfigurationFile**(*msg*, *do_log=True*)
Raised when the configuration file cannot be read (e.g., does not exist or has wrong permissions).

exception gc3libs.exceptions.**NoConfigurationFile**(*msg*, *do_log=True*)
Raised when the configuration file cannot be read (e.g., does not exist or has wrong permissions), or cannot be parsed (e.g., is malformed).

exception gc3libs.exceptions.**NoResources**(*msg*, *do_log=False*)
Raised to signal that no resources are defined, or that none are compatible with the request.

exception gc3libs.exceptions.**NoValidConfigurationFile**(*msg*, *do_log=True*)
Raised when the configuration file cannot be parsed (e.g., is malformed).

exception gc3libs.exceptions.**OutputNotAvailableError**(*msg*, *do_log=False*)
Raised upon attempts to retrieve the output for jobs that are still in *NEW* or *SUBMITTED* state.

exception gc3libs.exceptions.**RecoverableDataStagingError**(*msg*, *do_log=False*)
Raised when transient problems with copying data to or from the remote execution site occurred.

This error is considered to be transient (e.g., network connectivity interruption), so trying again at a later time could solve the problem.

exception gc3libs.exceptions.**RecoverableError**(*msg*, *do_log=False*)
Used to mark transient errors: retrying the same action at a later time could succeed.

This exception should *never* be instanciated: it is only to be used in *except* clauses to catch "try again" situations.

exception gc3libs.exceptions.**TaskError**(*msg*, *do_log=False*)
Generic error condition in a *Task* object.

exception gc3libs.exceptions.**UnexpectedStateError**(*msg*, *do_log=False*)
Raised by `Task.progress()` when a job lands in *STOPPED* or *TERMINATED* state.

exception gc3libs.exceptions.**UnknownJob**(*msg*, *do_log=False*)
Raised when an operation is attempted on a task, which is unknown to the remote server or backend.

exception gc3libs.exceptions.**UnknownJobState**(*msg*, *do_log=False*)
Raised when a job state is gotten from the Grid middleware, that is not handled by the GC3Libs code. Might actually mean that there is a version mismatch between GC3Libs and the Grid middleware used.

exception gc3libs.exceptions.**UnrecoverableDataStagingError**(*msg*, *do_log=False*)
Raised when problems with copying data to or from the remote execution site occurred.

exception gc3libs.exceptions.**UnrecoverableError**(*msg*, *do_log=False*)
Used to mark permanent errors: there's no point in retrying the same action at a later time, because it will yield the same error again.

This exception should *never* be instanciated: it is only to be used in *except* clauses to exclude "try again" situations.

gc3libs.optimizer

Support for finding minima of functions with GC3Pie.

GC3Pie can run a large number of *Application* instances in parallel. The idea of this optimization module is to use these core capabilities to perform optimization, which is particularly effective for optimization using evolutionary algorithms, as they require several independent evaluations of the target function.

The optimization module has two main components, the driver and the algorithm. You need both an instance of a driver and an instance of an algorithm to perform optimization of a given function.

Drivers perform optimization following a specific algorithm. Two drivers are currently implemented: *drivers.SequentialDriver* that runs the entire algorithm on the local computer (hence, all the evaluations of the target function required by the algorithm are performed one after the other), and *drivers.ParallelDriver* splits the evaluations into tasks that are executed in parallel using GC3Pie's remote execution facilities.

This module implements a generic framework for evolutionary algorithms, and one particular type of global optimization algorithm called Differential Evolution is worked out in full. Other Evolutionary Algorithms can easily be incorporated by subclassing *EvolutionaryAlgorithm*. (Different optimization algorithms, for example gradient based methods such as quasi-newton methods, could be implemented but likely require adaptations in the driver classes.)

The module is organized as follows:

- *drivers*: Set of drivers that interface with GC3Libs to automatically drive the optimization process following a specified algorithm. *ParallelDriver* is the core of the optimization module, performing optimization using an algorithm based on *EvolutionaryAlgorithm*.
- *dif_evolution*: Implements the Differential Evolution algorithm, in particular the evolution and selection step, based on *EvolutionaryAlgorithm*. See the module for details on the algorithm.
- *extra*: Provides tools to printing, plotting etc. that can be used as addons to *EvolutionaryAlgorithm*.

class gc3libs.optimizer.**EvolutionaryAlgorithm**(*initial_pop*, *itermax=100*, *dx_conv_crit=None*, *y_conv_crit=None*, *logger=None*, *after_update_opt_state=[]*)
Base class for building an evolutionary algorithm for global optimization.

Parameters

- **initial_pop** (list) – Initial population for the optimization.
- **itermax** (*int*) – Maximum # of iterations.
- **dx_conv_crit** (*float*) – Abort optimization if all population members are within a certain distance to each other.
- **y_conv_crit** (*float*) – Declare convergence when the target function is below a *y_conv_crit*.
- **logger** (*obj*) – Configured logger to use.
- **after_update_opt_state** (list) – Functions that are called at the end of *update_opt_state()*. Use this list to provide problem-specific printing and plotting routines. Examples can be found in *gc3libs.optimizer.extra*.

evolve()
Generates a new population fullfilling in_domain(). :rtype list of population members

has_converged()
Checks convergence based on two criteria:

1. Is the lowest target value in the population below *y_conv_crit*.
2. Are all population members within *dx_conv_crit* from the first population member.

Return type bool

select(*new_pop*, *new_vals*)
Update *self.pop* and *self.vals* given the new population and the corresponding fitness vector.

update_opt_state(*new_pop*, *new_vals*)
Stores set of function values corresponding to the current population, then updates optimizer state in many ways:

- update the *.best** variables accordingly;
- uses *select()* to determine the surviving population.
- advances iteration count.

gc3libs.optimizer.**draw_population**(*lower_bds*, *upper_bds*, *dim*, *size*, *in_domain=None*, *seed=None*)
Draw a random population with the following criteria:

Parameters

- **lower_bds** (list) – List of length *dim* indicating the lower bound in each dimension.
- **upper_bds** (list) – List of length *dim* indicating the upper bound in each dimension.
- **dim** (*int*) – Dimension of each population member.
- **size** (*int*) – Population size.
- **in_domain** (*fun*) – Determines population's validity. Takes no arguments and returns a list of bools indicating each members validity.
- **seed** (*float*) – Seed to initialize NumPy's random number generator.

Return type list of population members

gc3libs.optimizer.**populate**(*create_fn*, *in_domain=None*, *max_n_resample=100*)
Uses create_fn() to generate a new population. If in_domain() is not fulfilled, create_fn() is called repeatedly. Invalid population members are replaced until reaching the desired valid population size or *max_n_resample* calls to create_fn(). If *max_n_resample* is reached, a warning is issued and the optimization continues with the remaining "invalid" members.

Parameters

- **create_fn** (*fun*) – Generates a new population. Takes no arguments.
- **in_domain** (*fun*) – Determines population's validity. Takes no arguments and returns a list of bools indicating each members validity.
- **max_n_resample** (*int*) – Maximum number of resamples to be drawn to satisfy :func:'in_domain

Return type list of population members

gc3libs.optimizer.dif_evolution

This module implements a global optimization algorithm called Differential Evolution.

Consider the following optimization problem: $min\ f(\mathbf{x})\ \ s.t.\ \ \mathbf{x} \in D$, where $D \in R^d$ and $f : D \mapsto R$. Class `DifferentialEvolutionAlgorithm` solves this optimization problem using the differential evolution algorithm. No further assumptions on the function f are needed. Thus it can be non-convex, noisy etc.

The domain D is implicitly specified by passing the function `filtern_fn()` to `DifferentialEvolutionAlgorithm`.

Some information related to Differential Evolution can be found in the following papers:

1. Tvrdik 2008: http://www.proceedings2008.imcsit.org/pliks/95.pdf
2. Fleetwood: http://www.maths.uq.edu.au/MASCOS/Multi-Agent04/Fleetwood.pdf
3. Piyasatian: http://www-personal.une.edu.au/~jvanderw/DE_1.pdf

`evolve_fn()` is an adaptation of the following MATLAB code: http://www.icsi.berkeley.edu/~storn/DeMat.zip hosted on http://www.icsi.berkeley.edu/~storn/code.html#deb1.

class `gc3libs.optimizer.dif_evolution.`**DifferentialEvolutionAlgorithm**(*initial_pop*, *de_strategy='DE_rand'*, *de_step_size=0.85*, *prob_crossover=1.0*, *exp_cross=False*, *itermax=100*, *dx_conv_crit=None*, *y_conv_crit=None*, *in_domain=None*, *seed=None*, *logger=None*, *after_update_opt_state=[]*)

Differential Evolution Algorithm class. `DifferentialEvolutionAlgorithm` explicitly allows for an another process to control the optimization. Driver classes can be found in `gc3libs.optimizer.drivers.py`.

Parameters

- **initial_pop** (list) – Initial population for the optimization.
- **de_strategy** (*str*) – e.g. DE_rand_either_or_algorithm. Allowed are:
- **de_step_size** (*float*) – Differential Evolution step size.
- **prob_crossover** (*float*) – Probability new population draws will replace old members.
- **exp_cross** (*bool*) – Set True to use exponential crossover.
- **itermax** (*int*) – Maximum # of iterations.

- **dx_conv_crit** (*float*) – Abort optimization if all population members are within a certain distance to each other.
- **y_conv_crit** (*float*) – Declare convergence when the target function is below a *y_conv_crit*.
- **in_domain** (*fun*) – Optional function that implements nonlinear constraints.
- **seed** (*float*) – Seed to initialize NumPy's random number generator.
- **logger** (*obj*) – Configured logger to use.
- **after_update_opt_state** (list) – Functions that are called at the end of `DifferentialEvolutionAlgorithm.after_update_opt_state()`. Use this list to provide problem-specific printing and plotting routines. Examples can be found in *gc3libs.optimizer.extra*.

The *de_strategy* value must be chosen from the *dif_evolution.strategies* enumeration. Allowed values are (description of the strategies taken from http://www.icsi.berkeley.edu/~storn/DeMat.zip):

1. `'DE_rand'`: The classical version of DE.
2. **`'DE_local_to_best'`: A version which has been used by quite a number of** scientists. Attempts a balance between robustness # and fast convergence.
3. **`'DE_best_with_jitter'`: Taylored for small population sizes and fast** convergence. Dimensionality should not be too high.
4. `'DE_rand_with_per_vector_dither'`: Classical DE with dither to become even more robust.
5. **`'DE_rand_with_per_generation_dither'`: Classical DE with dither to become even more robust.** Choosing de_step_size = 0.3 is a good start here.
6. `'DE_rand_either_or_algorithm'`: Alternates between differential mutation and three-point-recombination.

evolve()
Generates a new population fullfilling *in_domain*.

Return type list of population members

static evolve_fn(*population*, *prob_crossover*, *de_step_size*, *dim*, *best_iter*, *de_strategy*, *exp_cross*)
Return new population, evolved according to *de_strategy*.

Parameters

- **population** – Population generating offspring from.
- **prob_crossover** – Probability new population draws will replace old members.
- **de_step_size** – Differential Evolution step size.
- **dim** – Dimension of each population member.
- **best_iter** – Best population member of the current population.
- **de_strategy** – Differential Evolution strategy. See *DifferentialEvolutionAlgorithm*.
- **bool** (*exp_cross*) – Set True to use exponential crossover.

select(*new_pop*, *new_vals*)
Perform a one-on-one battle by index, keeping the member with lowest corresponding value.

gc3libs.optimizer.drivers

Drivers to perform global optimization.

Global optimizations can be performed sequentially on a local machine using *SequentialDriver*. To make use of parallelization, *ParallelDriver* allows submission of jobs to gc3pie ressources.

Drivers use an algorithm instance that conforms to *optimizer.EvolutionaryAlgorithm* to generate new populations.

class gc3libs.optimizer.drivers.**ComputeTargetVals**(*pop*, *jobname*, *iteration*, *path_to_stage_dir*, *cur_pop_file*, *task_constructor*, ***extra_args*)

gc3libs.workflow.ParallelTaskCollection to evaluate the current *pop* using the user-supplied `task_constructor()`.

Parameters

- **pop** (list) – Population to evaluate.
- **jobname** (*str*) – Name of `GridDriver` instance driving the optimization.
- **iteration** (*int*) – Current iteration number.
- **path_to_stage_dir** (*str*) – Path to directory in which optimization takes place.
- **cur_pop_file** (*str*) – Filename under which the population is stored in the

current iteration dir. The population is discarded if no file is specified. :param task_constructor: Takes a list of x vectors and the path to the current iteration directory. Returns Application instances that can be executed on the grid.

class gc3libs.optimizer.drivers.**ParallelDriver**(*jobname=''*, *path_to_stage_dir=''*, *opt_algorithm=None*, *task_constructor=None*, *extract_value_fn=<function <lambda>>*, *cur_pop_file=''*, ***extra_args*)

Drives an optimization using *opt_algorithm* on the grid.

At each iteration an instance of *ComputeTargetVals* uses `task_constructor()` to generate *gc3libs.Application* instances to be executed in parallel. When all jobs are complete, the output is analyzed with the user-supplied function `extract_value_fn()`. This function returns the function value for all analyzed input vectors.

Parameters

- **jobname** (*str*) – string that labels this optimization case.
- **path_to_stage_dir** – directory in which to perform the optimization.
- **opt_algorithm** – Evolutionary algorithm instance that conforms to *optimizer.EvolutionaryAlgorithm*.
- **task_constructor** – A function that takes a list of x vectors and the path to the current iteration directory, and returns Application instances that can be executed on the grid.
- **extract_value_fn** – Takes an `Application` instance returns the function value computed in that task. The default implementation just looks for a *.value* attribute on the application instance.
- **cur_pop_file** – Filename under which the population is stored in the current iteration dir. The population is discarded if no file is specified.

Optimization drivers use GC3Pie in the following way: A *SequentialTaskCollection* represents the main loop of the optimization algorithm, checking for convergence at each iteration. This allows for resuming paused or crashed optimizations. Each iteration, the optimization algorithm provides a new set of points to be evaluated. These points are each represented by an *Application* and bundled into a

ParallelTaskCollection that manages each single *Application* until completion. The structure of GC3Libs objects employed can be summarized as follows:

```
SequentialTaskCollection
          |
          v
  ParallelTaskCollection
          |
          v
    Application
```

class gc3libs.optimizer.drivers.**SequentialDriver**(*opt_algorithm*, *target_fn*, *path_to_stage_dir='/home/docs/checkouts/readthedocs.org/u*, *cur_pop_file=None*, *logger=None*, *fmt=None*)

Drives an optimization using *opt_algorithm* on the local machine.

The user-supplied `target_fun()` computes target values for the populations generated by *opt_algorithm*.

Parameters

- **opt_algorithm** – Evolutionary algorithm instance that conforms to *optimizer.EvolutionaryAlgorithm*.
- **target_fn** – Function to evaluate a population and return the corresponding values.
- **path_to_stage_dir** – Directory in which to perform the optimization.
- **cur_pop_file** – Filename under which the population is stored in the current iteration dir. The population is discarded if no file is specified.
- **logger** – Configured logger to use.
- **fmt** (*str*) – `%`-format string to use (e.g., `%12.8f`) to print values at each step of the algorithm. If *None* (default), this verbose report is not generated, as it might be time-consuming for large population sizes.

de_opt()

Drives optimization until convergence or *itermax* is reached.

gc3libs.optimizer.extra

Collection of tools to supplement optimization algorithm *optimizer.EvolutionaryAlgorithm*.

Include a list of desired tools in param *after_update_opt_state* of *optimizer.EvolutionaryAlgorithm*.

gc3libs.optimizer.extra.**log_stats**(*algo*, *logger=<logging.RootLogger object>*)

Log summary statistics for *algo*.

Parameters **algo** (*str*) – Instance of *gc3libs.optimizer.EvolutionaryAlgorithm*.

class gc3libs.optimizer.extra.**plot_population**(*figure_dir*)

Plot the 2-dimensional population of an *gc3libs.optimizer.EvolutionaryAlgorithm* instance. If the population is not 2-d an error message appears and no plot is created.

Parameters **figure_dir** (*str*) – Path to the directory where plots should be stored. Directory will be created if non-existent.

gc3libs.optimizer.extra.**print_stats**(*algo*, *output=<open file '<stdout>', mode 'w'>*)

Print summary statistics for *algo*.

Parameters

- **algo** (*str*) – Instance of *gc3libs.optimizer.EvolutionaryAlgorithm*.
- **output** – Output stream.

gc3libs.persistence

Facade to store and retrieve Job information from permanent storage.

A usage warning

This module saves Python objects using the *pickle* framework: thus, the *Application* subclass corresponding to a job must be already loaded (or at least `import`-able) in the Python interpreter for *pickle* to be able to 'undump' the object from its on-disk representation.

In other words, if you create a custom *Application* subclass in some client code, GC3Utils won't be able to read job files created by this code, because the class definition is not available in GC3Utils.

The recommended simple workaround is for a stand-alone script to 'import self' and then use the fully qualified name to run the script. In other words, start your script with this boilerplate code:

```
if __name__ == '__main__':
    import myscriptname
    myscriptname.MyScript().run()
```

The rest of the script now runs as the `myscript` module, which does the trick!

Note: Of course, the `myscript.py` file must be in the search path of the Python interpreter, or GC3Utils will still complain!

`gc3libs.persistence.`**`make_store`**(*uri*, **args*, ***extra_args*)

Factory producing concrete `Store` instances.

Given a URL and (optionally) initialization arguments, return a fully-constructed `Store` instance.

The only required argument is *uri*; if any other arguments are present in the function invocation, they are passed verbatim to the constructor associated with the scheme of the given *uri*.

Example:

```
>>> fs1 = make_store('file:///tmp')
>>> fs1.__class__.__name__
'FilesystemStore'
```

Argument *uri* can also consist of a path name, in which case a URL scheme 'file:///' is assumed:

```
>>> fs2 = make_store('/tmp')
>>> fs2.__class__.__name__
'FilesystemStore'
```

class `gc3libs.persistence.`**`Persistable`**(**args*, ***kwargs*)

A mix-in class to mark that an object should be persisted by its ID.

Any instance of this class is saved as an 'external reference' when a container holding a reference to it is saved.

class `gc3libs.persistence.`**`IdFactory`**(*prefix=None*, *next_id_fn=None*, *id_class=<class 'gc3libs.persistence.idfactory.Id'>*)

Automatically generate a "unique identifier" (of class *Id*). Object identifiers are temporally unique: no identifier will (ever) be re-used, even in different invocations of the program.

`new`(*obj*)

Return a new "unique identifier" instance (a string).

`reserve`(*n*)

Pre-allocate *n* IDs. Successive invocations of the *Id* constructor will return one of the pre-allocated, with a potential speed gain if many *Id* objects are constructed in a loop.

class `gc3libs.persistence.`**`JobIdFactory`**(*next_id_fn=None*)

Override *`IdFactory`* behavior and generate IDs starting with a lowercase `job` prefix.

class gc3libs.persistence.**FilesystemStore**(*directory='/home/docs/.gc3/jobs'*, *idfactory=<gc3libs.persistence.idfactory.IdFactory object>*, *protocol=2*, ***extra_args*)

Save and load objects in a given directory. Uses Python's standard *pickle* module to serialize objects onto files.

All objects are saved as files in the given directory (default: *gc3libs.Default.JOBS_DIR*). The file name is the object ID.

If an object contains references to other *Persistable* objects, these are saved in the file they would have been saved if the *save* method was called on them in the first place, and only an 'external reference' is saved in the pickled container. This ensures that: (1) only one copy of a shared object is ever saved, and (2) any shared reference to *Persistable* objects is correctly restored when restoring the container.

The default *idfactory* assigns object IDs by appending a sequential number to the class name; see class *Id* for details.

The *protocol* argument specifies the serialization protocol to use, if different from *gc3libs.persistence.serialization.DEFAULT_PROTOCOL*.

Any extra keyword arguments are ignored for compatibility with *SqlStore*.

list()
Return list of IDs of saved *Job* objects.

This is an optional method; classes that do not implement it should raise a *NotImplementedError* exception.

load(*id_*)
Load a saved object given its ID, and return it.

remove(*id_*)
Delete a given object from persistent storage, given its ID.

replace(*id_*, *obj*)
Replace the object already saved with the given ID with a copy of *obj*.

save(*obj*)
Save an object, and return an ID.

gc3libs.persistence.accessors

Accessors for object attributes and container items.

gc3libs.persistence.accessors.**GET = <gc3libs.persistence.accessors.GetValue object>**
Constant identity getter.

Use this for better readability (e.g., *GET[0]* instead of *GetValue()[0]*).

class gc3libs.persistence.accessors.**GetAttributeValue**(*attr*, *xform=<function <lambda>>*, *default=<object object>*)

Return an accessor function for the given attribute.

An instance of *GetAttributeValue* is a callable that, given any object, returns the value of its attribute *attr*, whose name is specified in the *GetAttributeValue* constructor:

```
>>> from gc3libs import Struct
 >>> fn = GetAttributeValue('x')
 >>> a = Struct(x=1, y=2)
 >>> fn(a)
 1
```

The accessor raises *AttributeError* if no such attribute exists):

```
>>> b = Struct(z=3)
>>> fn(b)
Traceback (most recent call last):
   ...
AttributeError: 'Struct' object has no attribute 'x'
```

However, you can specify a default value, in which case the default value is returned and no error is raised:

```
>>> fn = GetAttributeValue('x', default=42)
>>> fn(b)
42
>>> fn = GetAttributeValue('y', default=None)
>>> print(fn(b))
None
```

In other words, if *fn = GetAttributeValue('x')*, then *fn(obj)* evaluates to *obj.x*.

If the string *attr* contains any dots, then attribute lookups are chained: if *fn = GetAttributeValue('x.y')* then *fn(obj)* evaluates to *obj.x.y*:

```
>>> fn = GetAttributeValue('x.y')
>>> a = Struct(x=Struct(y=42))
>>> fn(a)
42
```

The optional second argument *xform* allows composing the accessor with an arbitrary function that is passed an object and should return a (possibly different) object whose attributes should be looked up. In other words, if *xform* is specified, then the returned accessor function computes *xform(obj).attr* instead of *obj.attr*.

This allows combining *GetAttributeValue* with *GetItemValue()* (which see), to access objects in deeply-nested data structures; see *GetItemValue* for examples.

class gc3libs.persistence.accessors.**GetItemValue**(*place*, *xform=<function <lambda>>*, *default=<object object>*)

Return accessor function for the given item in a sequence.

An instance of *GetItemValue* is a callable that, given any sequence/container object, returns the value of the item at its place *idx*:

```
>>> fn = GetItemValue(1)
>>> a = 'abc'
>>> fn(a)
'b'
>>> b = { 1:'x', 2:'y' }
>>> fn(b)
'x'
```

In other words, if *fn = GetItemValue(x)*, then *fn(obj)* evaluates to *obj[x]*.

Note that the returned function *fn* raises *IndexError* or *KeyError*, (depending on the type of sequence/container) if place *idx* does not exist:

```
>>> fn = GetItemValue(42)
>>> a = list('abc')
>>> fn(a)
Traceback (most recent call last):
   ...
IndexError: list index out of range
>>> b = dict(x=1, y=2, z=3)
>>> fn(b)
Traceback (most recent call last):
   ...
KeyError: 42
```

However, you can specify a default value, in which case the default value is returned and no error is raised:

```
>>> fn = GetItemValue(42, default='foo')
>>> fn(a)
'foo'
>>> fn(b)
'foo'
```

The optional second argument *xform* allows composing the accessor with an arbitrary function that is passed an object and should return a (possibly different) object where the item lookup should be performed. In other words, if *xform* is specified, then the returned accessor function computes *xform(obj)[idx]* instead of *obj[idx]*. For example:

```
>>> c = 'abc'
>>> fn = GetItemValue(1, xform=(lambda s: s.upper()))
>>> fn(c)
'B'

>>> c = (('a',1), ('b',2))
>>> fn = GetItemValue('a', xform=dict)
>>> fn(c)
1
```

This allows combining *GetItemValue* with `GetAttrValue` (which see), to access objects in deeply-nested data structures.

class gc3libs.persistence.accessors.**GetOnly**(*only*, *xform=<function <lambda>>*, *default=<object object>*)

Apply accessor function to members of a certain class; return a default value otherwise.

The *GetOnly* accessor performs just like *GetValue*, but is effective only on instances of a certain class; if the accessor function is passed an instance of a different class, the default value is returned:

```
>>> from gc3libs import Struct
>>> fn4 = GetOnly(Struct, default=42)
>>> isinstance(fn4(Struct(foo='bar')), Struct)
True
>>> isinstance(fn4(dict(foo='bar')), dict)
False
>>> fn4(dict(foo='bar'))
42
```

If *default* is not specified, then *None* is returned:

```
>>> fn5 = GetOnly(Struct)
>>> repr(fn5(dict(foo='bar')))
'None'
```

class gc3libs.persistence.accessors.**GetValue**(*default=<object object>*)

Provide easier compositional syntax for *GetAttributeValue* and *GetItemValue*.

Instances of *GetAttributeValue* and *GetItemValue* can be composed by passing one as *xform* parameter to the other; however, this results in the writing order being the opposite of the composition order: for instance, to create an accessor to evaluate *x.a[0]* for any Python object *x*, one has to write:

```
>>> from gc3libs import Struct
>>> fn1 = GetItemValue(0, GetAttributeValue('a'))
```

The *GetValue* class allows to write accessor expressions the way they are normally written in Python:

```
>>> GET = GetValue()
>>> fn2 = GET.a[0]
>>> x = Struct(a=[21,42], b='foo')
>>> fn1(x)
21
>>> fn2(x)
21
```

The optional *default* argument specifies a value that should be used in case the required attribute or item is not found:

```
>>> fn3 = GetValue(default='no value found').a[3]
>>> fn3(x)
'no value found'
```

ONLY(*specifier*)
Restrict the action of the accessor expression to members of a certain class; return default value otherwise.

The invocation to `only()` should *always be last*:

```
>>> from gc3libs import Struct
>>> fn = GetValue(default='foo').a[0].ONLY(Struct)
>>> fn(Struct(a=['bar','baz']))
'bar'
>>> fn(dict(a=['bar','baz']))
'foo'
```

If it's not last, you will get *AttributeError* like the following:

```
>>> fn = GetValue().ONLY(Struct).a[0]
>>> fn(dict(a=[0,1]))
Traceback (most recent call last):
  ...
AttributeError: 'NoneType' object has no attribute 'a'
```

gc3libs.persistence.filesystem

class gc3libs.persistence.filesystem.**FilesystemStore**(*directory='/home/docs/.gc3/jobs'*, *idfactory=<gc3libs.persistence.idfactory.IdFactory object>*, *protocol=2*, ***extra_args*)
Save and load objects in a given directory. Uses Python's standard *pickle* module to serialize objects onto files.

All objects are saved as files in the given directory (default: *gc3libs.Default.JOBS_DIR*). The file name is the object ID.

If an object contains references to other *Persistable* objects, these are saved in the file they would have been saved if the *save* method was called on them in the first place, and only an 'external reference' is saved in the pickled container. This ensures that: (1) only one copy of a shared object is ever saved, and (2) any shared reference to *Persistable* objects is correctly restored when restoring the container.

The default *idfactory* assigns object IDs by appending a sequential number to the class name; see class *Id* for details.

The *protocol* argument specifies the serialization protocol to use, if different from *gc3libs.persistence.serialization.DEFAULT_PROTOCOL*.

Any extra keyword arguments are ignored for compatibility with *SqlStore*.

list()
Return list of IDs of saved *Job* objects.

This is an optional method; classes that do not implement it should raise a *NotImplementedError* exception.

load(*id_*)
Load a saved object given its ID, and return it.

remove(*id_*)
Delete a given object from persistent storage, given its ID.

replace(*id_*, *obj*)
Replace the object already saved with the given ID with a copy of *obj*.

save(*obj*)
Save an object, and return an ID.

gc3libs.persistence.filesystem.**make_filesystemstore**(*url*, **args*, ***extra_args*)
Return a *FilesystemStore* instance, given a 'file:///' URL and optional initialization arguments.

This function is a bridge between the generic factory functions provided by *gc3libs.persistence.make_store()* and gc3libs.persistence.register() and the class constructor *FilesystemStore*:class.

Examples:

```
>>> fs1 = make_filesystemstore(gc3libs.url.Url('file:///tmp'))
>>> fs1.__class__.__name__
'FilesystemStore'
```

gc3libs.persistence.idfactory

class gc3libs.persistence.idfactory.**Id**
An automatically-generated "unique identifier" (a string-like object). The unique object identifier has the form "PREFIX.NNN" where "NNN" is a decimal number, and "PREFIX" defaults to the object class name but can be overridden in the *Id* constructor.

Two object IDs can be compared iff they have the same prefix; in which case, the result of the comparison is the same as comparing the two sequence numbers.

class gc3libs.persistence.idfactory.**IdFactory**(*prefix=None*, *next_id_fn=None*, *id_class=<class 'gc3libs.persistence.idfactory.Id'>*)
Automatically generate a "unique identifier" (of class *Id*). Object identifiers are temporally unique: no identifier will (ever) be re-used, even in different invocations of the program.

new(*obj*)
Return a new "unique identifier" instance (a string).

reserve(*n*)
Pre-allocate *n* IDs. Successive invocations of the *Id* constructor will return one of the pre-allocated, with a potential speed gain if many *Id* objects are constructed in a loop.

class gc3libs.persistence.idfactory.**JobIdFactory**(*next_id_fn=None*)
Override *IdFactory* behavior and generate IDs starting with a lowercase job prefix.

gc3libs.persistence.serialization

Generic object serialization (using Python's *pickle*/*cPickle* modules).

See the documentation for Python's standard *pickle* and *cPickle* modules for more details.

gc3libs.persistence.sql

SQL-based storage of GC3pie objects.

class gc3libs.persistence.sql.**SqlStore**(*url*, *table_name='store'*, *idfactory=None*, *extra_fields={}*, *create=True*, ***extra_args*)
Save and load objects in a SQL db, using python's *pickle* module to serialize objects into a specific field.

Access to the DB is done via SQLAlchemy module, therefore any driver supported by SQLAlchemy will be supported by this class.

The *url* argument is used to access the store. It is supposed to be a *gc3libs.url.Url* class, and therefore may contain username, password, host and port if they are needed by the db used.

The *table_name* argument is the name of the table to create. By default it's `store`.

The constructor will create the *table_name* table if it does not exist, but if there already is such a table it will assume the it's schema is compatible with our needs. A minimal table schema is as follow:

The meaning of the fields is:

id: this is the id returned by the *save()* method and univoquely identify a stored object.

data: the serialization of the object.

state: if the object is a *Task* istance this wil lbe the current execution state of the job

Field	Type	Null	Key	Default
id data state	int(11) blob varchar(128)	NO YES YES	PRI	NULL NULL NULL

The *extra_fields* argument is used to extend the database. It must contain a mapping <*column*> : <*function*> where:

<*column*> is a *sqlalchemy.Column* object.

<*function*> is a function which takes the object to be saved as argument and returns the value to be stored into the database. Any exception raised by this function will be *ignored*. Classes `GetAttribute` and `GetItem` in module `get` provide convenient helpers to save object attributes into table columns.

For each extra column the *save()* method will call the corresponding <*function*> in order to get the correct value to store into the db.

Any extra keyword arguments are ignored for compatibility with *FilesystemStore*.

list()
Return list of IDs of saved *Job* objects.

This is an optional method; classes that do not implement it should raise a *NotImplementedError* exception.

load(*id_*)
Load a saved object given its ID, and return it.

remove(*id_*)
Delete a given object from persistent storage, given its ID.

replace(*id_*, *obj*)
Replace the object already saved with the given ID with a copy of *obj*.

save(*obj*)
Save an object, and return an ID.

gc3libs.persistence.sql.**make_sqlstore**(*url*, **args*, ***extra_args*)
Return a *SqlStore* instance, given a SQLAlchemy URL and optional initialization arguments.

This function is a bridge between the generic factory functions provided by *gc3libs.persistence.make_store()* and `gc3libs.persistence.register()` and the class constructor *SqlStore*:class.

Examples:

```
| >>> ss1 = make_sqlstore(gc3libs.url.Url('sqlite:////tmp/foo.db'))
| >>> ss1.__class__.__name__
| 'SqlStore'
```

gc3libs.persistence.sql.**sql_next_id_factory**(*db*)
This function will return a function which can be used as *next_id_fn* argument for the *IdFactory* class constructor.

db is DB connection class conform to DB API2.0 specs (works also with SQLAlchemy engine types)

The function returned has signature:

sql_next_id(n=1)

the id returned is the maximum *id* field in the *store* table plus 1.

gc3libs.persistence.store

class gc3libs.persistence.store.**Persistable**(**args*, ***kwargs*)
A mix-in class to mark that an object should be persisted by its ID.

Any instance of this class is saved as an 'external reference' when a container holding a reference to it is saved.

class gc3libs.persistence.store.**Store**
Interface for storing and retrieving objects on permanent storage.

Each *save* operation returns a unique "ID"; each ID is a Python string value, which is guaranteed to be temporally unique, i.e., no two *save* operations in the same persistent store can result in the same IDs being assigned to different objects. The "ID" is also stored in the instance attribute *_id*.

Any Python object can stored, provided it meets the following conditions:

- it can be pickled with Python's standard module *pickle*.
- the instance attribute *persistent_id* is reserved for use by the *Store* class: it should not be set or altered by other parts of the code.

list(***extra_args*)
Return list of IDs of saved *Job* objects.

This is an optional method; classes that do not implement it should raise a *NotImplementedError* exception.

load(*id_*)
Load a saved object given its ID, and return it.

remove(*id_*)
Delete a given object from persistent storage, given its ID.

replace(*id_*, *obj*)
Replace the object already saved with the given ID with a copy of *obj*.

save(*obj*)
Save an object, and return an ID.

gc3libs.persistence.store.**make_store**(*uri*, **args*, ***extra_args*)
Factory producing concrete *Store* instances.

Given a URL and (optionally) initialization arguments, return a fully-constructed *Store* instance.

The only required argument is *uri*; if any other arguments are present in the function invocation, they are passed verbatim to the constructor associated with the scheme of the given *uri*.

Example:

```
>>> fs1 = make_store('file:///tmp')
>>> fs1.__class__.__name__
'FilesystemStore'
```

Argument *uri* can also consist of a path name, in which case a URL scheme 'file:///' is assumed:

```
>>> fs2 = make_store('/tmp')
>>> fs2.__class__.__name__
'FilesystemStore'
```

gc3libs.persistence.store.**register**(*scheme*, *constructor*)
Register *constructor* as the factory corresponding to an URL scheme.

If a different constructor is already registered for the same scheme, it is silently overwritten.

The registry mapping schemes to constructors is used in the *make_store()* to create concrete instances of *gc3libs.persistence.Store*, given a URI that identifies the kind and location of the storage.

Parameters

- **scheme** (*str*) – URL scheme to associate with the given constructor.
- **constructor** (*callable*) – A callable returning a *Store*

instance. Typically, a class constructor.

gc3libs.quantity

Manipulation of quantities with units attached with automated conversion among compatible units.

For details and the discussion leading up to this, see: *<http://code.google.com/p/gc3pie/issues/detail?id=47>*

class gc3libs.quantity.**Duration**
Represent the duration of a time lapse.

Construction of a duration can be done by parsing a string specification; several formats are accepted:

•A duration is an aggregate of days, hours, minutes and seconds:

```
>>> l3 = Duration('1day 4hours 9minutes 16seconds')
>>> l3.amount(Duration.s) # convert to seconds
101356
```

•Any of the terms can be omitted (in which case it defaults to zero):

```
>>> l4 = Duration('1day 4hours 16seconds')
>>> l4 == l3 - Duration('9 minutes')
True
```

•The unit names can be singular or plural, and any amount of space can be added between the time unit name and the associated amount:

```
>>> l5 = Duration('3 hour 42 minute')
>>> l6 = Duration('3 hours 42 minutes')
>>> l7 = Duration('3hours 42minutes')
>>> l5 == l6 == l7
True
```

•Unit names can also be abbreviated using just the leading letter:

```
>>> l8 = Duration('3h 42m')
>>> l9 = Duration('3h42m')
>>> l8 == l9
True
```

•The abbreviated formats HH:MM:SS and DD:HH:MM:SS are also accepted:

```
>>> # 1 hour + 1 minute + 1 second
>>> l1 = Duration('01:01:01')
>>> l1 == Duration('3661 s')
True

>>> # 1 day, 2 hours, 3 minutes, 4 seconds
>>> l2 = Duration('01:02:03:04')
>>> l2.amount(Duration.s)
93784
```

However, the formats HH:MM and MM:SS are rejected as ambiguous:

```
>>> # is this hours:minutes or minutes:seconds ?
>>> l0 = Duration('01:02')
Traceback (most recent call last):
  ...
ValueError: Duration '01:02' is ambiguous: use '1m 2s' ...
```

•Finally, you can specify a duration like any other quantity, as an integral amount of a given time unit:

```
>>> l1 = Duration('1 day')
>>> l2 = Duration('86400 s')
>>> l1 == l2
True
```

A new quantity can also be defined as a multiple of an existing one:

```
>>> an_hour = Duration('1 hour')
>>> a_day = 24 * an_hour
>>> a_day.amount(Duration.h)
24
```

The quantities `Duration.hours`, `Duration.minutes` and `Duration.seconds` (and their single-letter abbreviations `h`, `m`, `s`) are pre-defined with their obvious meaning.

Also module-level aliases `hours`, `minutes` and `seconds` (and the one-letter forms) are available:

```
>>> a_day1 = 24*hours
>>> a_day2 = 1440*minutes
>>> a_day3 = 86400*seconds
```

This allows for yet another way of constructing duration objects, i.e., by passing the amount and the unit separately to the constructor:

```
>>> a_day4 = Duration(24, hours)
```

Two durations are equal if they indicate the exact same amount in seconds:

```
>>> a_day1 == a_day2
True
>>> a_day1.amount(s)
86400
>>> a_day2.amount(s)
86400

>>> a_day == an_hour
False
>>> a_day.amount(minutes)
1440
>>> an_hour.amount(minutes)
60
```

Basic arithmetic is possible with durations:

```
>>> two_hours = an_hour + an_hour
>>> two_hours == 2*an_hour
True

>>> one_hour = two_hours - an_hour
>>> one_hour.amount(seconds)
3600
```

It is also possible to add duration quantities defined with different units; the result is naturally expressed in the smaller unit of the two:

```
>>> one_hour_and_half = an_hour + 30*minutes
>>> one_hour_and_half
Duration(90, unit=m)
```

Note that the two unit class and numeric amount are accessible through the *unit* and `amount()` attributes:

```
>>> one_hour_and_half.unit
Duration(1, unit=m)
>>> one_hour_and_half.amount()
90
```

The `amount()` method accepts an optional specification of an alternate unit to express the amount into:

```
>>> one_hour_and_half.amount(Duration.hours)
1
```

An optional *conv* argument is available to specify a numerical domain for conversion, in case the default integer arithmetic is not precise enough:

```
>>> one_hour_and_half.amount(Duration.hours, conv=float)
1.5
```

The `to_str()` method allows representing a duration as a string, and provides choice of the output format and unit. The format string should contain exactly two `%`-specifiers: the first one is used to format the numerical amount, and the second one to format the measurement unit name.

By default, the unit used originally for defining the quantity is used:

```
>>> an_hour.to_str('%d [%s]')
'1 [hour]'
```

This can be overridden by specifying an optional second argument *unit*:

```
>>> an_hour.to_str('%d [%s]', unit=Duration.m)
'60 [m]'
```

A third optional argument *conv* can set the numerical type to be used for conversion computations:

```
>>> an_hour.to_str('%.1f [%s]', unit=Duration.m, conv=float)
'60.0 [m]'
```

The default numerical type is *int*, which in particular implies that you get a null amount if the requested unit is larger than the quantity:

```
>>> an_hour.to_str('%d [%s]', unit=Duration.days)
'0 [days]'
```

Conversion to string uses the unit originally used for defining the quantity and the `%g%s` format:

```
>>> str(an_hour)
'1hour'
```

to_timedelta(*duration*)

Convert a duration into a Python *datetime.timedelta* object.

This is useful to operate on Python's *datetime.time* and *datetime.date* objects, which can be added or subtracted to *datetime.timedelta*.

class gc3libs.quantity.**Memory**

Represent an amount of RAM.

Construction of a memory quantity can be done by parsing a string specification (amount followed by unit):

```
>>> byte = Memory('1 B')
>>> kilobyte = Memory('1 kB')
```

A new quantity can also be defined as a multiple of an existing one:

```
>>> a_thousand_kB = 1000*kilobyte
```

The base-10 units (up to TB, Terabytes) and base-2 (up to TiB, TiBiBytes) are available as attributes of the *Memory* class. This allows for a third way of constructing quantity objects, i.e., by passing the amount and the unit separately to the constructor:

```
>>> a_megabyte = Memory(1, Memory.MB)
>>> a_mibibyte = Memory(1, Memory.MiB)

>>> a_gigabyte = 1*Memory.GB
>>> a_gibibyte = 1*Memory.GiB

>>> two_terabytes = 2*Memory.TB
>>> two_tibibytes = 2*Memory.TiB
```

Two memory quantities are equal if they indicate the exact same amount in bytes:

```
>>> kilobyte == 1000*byte
True
>>> a_megabyte == a_mibibyte
False
>>> a_megabyte < a_mibibyte
True
>>> a_megabyte > a_gigabyte
False
```

Basic arithmetic is possible with memory quantities:

```
>>> two_bytes = byte + byte
>>> two_bytes == 2*byte
True
>>> half_gigabyte = a_gigabyte / 2
>>> half_gigabyte
Memory(476.837, unit=MiB)
```

The ratio of two memory quantities is correctly computed as a pure (floating-point) number:

```
>>> a_gigabyte / a_megabyte
1000.0
```

It is also possible to add memory quantities defined with different units; the result is naturally expressed in the smaller unit of the two:

```
>>> one_gigabyte_and_half = 1*Memory.GB + 500*Memory.MB
>>> one_gigabyte_and_half
Memory(1500, unit=MB)
```

Note that the two unit class and numeric amount are accessible through the *unit* and `amount()` attributes:

```
>>> one_gigabyte_and_half.unit
Memory(1, unit=MB)
>>> one_gigabyte_and_half.amount()
1500
```

The `amount()` method accepts an optional specification of an alternate unit to express the amount into:

```
>>> one_gigabyte_and_half.amount(Memory.GB)
1
```

An optional *conv* argument is available to specify a numerical domain for conversion, in case the default integer arithmetic is not precise enough:

```
>>> one_gigabyte_and_half.amount(Memory.GB, conv=float)
1.5
```

The `to_str()` method allows representing a quantity as a string, and provides choice of the output format and unit. The format string should contain exactly two `%`-specifiers: the first one is used to format the numerical amount, and the second one to format the measurement unit name.

By default, the unit used originally for defining the quantity is used:

```
>>> a_megabyte.to_str('%d [%s]')
'1 [MB]'
```

This can be overridden by specifying an optional second argument *unit*:

```
>>> a_megabyte.to_str('%d [%s]', unit=Memory.kB)
'1000 [kB]'
```

A third optional argument *conv* can set the numerical type to be used for conversion computations:

```
>>> a_megabyte.to_str('%g%s', unit=Memory.GB, conv=float)
'0.001GB'
```

The default numerical type is *int*, which in particular implies that you get a null amount if the requested unit is larger than the quantity:

```
>>> a_megabyte.to_str('%g%s', unit=Memory.GB, conv=int)
'0GB'
```

Conversion to string uses the unit originally used for defining the quantity and the `%g%s` format:

```
>>> str(a_megabyte)
'1MB'
```

class `gc3libs.quantity.`**`Quantity`**(*base_unit_name*, ***other_units*)

Metaclass for creating quantity classes.

This factory creates subclasses of `_Quantity` and bootstraps the base unit.

The name of the base unit is given as argument to the metaclass instance:

```
>>> class Memory1(object):
...   __metaclass__ = Quantity('B')
...
>>> B = Memory1('1 B')
>>> print (2*B)
2B
```

Optional keyword arguments create additional units; the argument key gives the unit name, and its value gives the ratio of the new unit to the base unit. For example:

```
>>> class Memory2(object):
...   __metaclass__ = Quantity('B', kB=1000, MB=1000*1000)
...
>>> a_thousand_kB = Memory2('1000kB')
>>> MB = Memory2('1   MB')
>>> a_thousand_kB == MB
True
```

Note that the units (base and additional) are also available as class attributes for easier referencing in Python code:

```
>>> a_thousand_kB == Memory2.MB
True
```

gc3libs.session

class `gc3libs.session.`**`Session`**(*path*, *store_url=None*, *create=True*, ***extra_args*)

A 'session' is a persistent collection of tasks.

Tasks added to the session are persistently recorded using an instance of *gc3libs.persistence.Store*. Stores can be shared among different sessions: each session knows wich jobs it 'owns'.

A session is associated to a directory, which holds all the data releated to that session. Specifically, two files are always created in the session directory andused internally by this class:

- *index.txt*: contains a list of all job IDs associated with this session;
- *store.url*: its contents are the URL of the store to create (as would be passed to the *gc3libs.persistence.make_store* factory).

To only argument needed to instantiate a session is the *path* of the directory; the directory name will be used as the identifier of the session itself. For example, the following code creates a temporary directory and the two files mentioned above inside it:

```
>>> import tempfile; tmpdir = tempfile.mktemp(dir='.')
>>> session = Session(tmpdir)
>>> sorted(os.listdir(tmpdir))
['created', 'session_ids.txt', 'store.url']
```

When a *Session* object is created with a *path* argument pointing to an existing valid session, the index of jobs is automatically loaded into memory, and the store pointed to by the `store.url` file in the session directory will be used, *disregarding the contents of the 'store_url' argument.*

In other words, the *store_url* argument is only used when *creating a new session*. If no *store_url* argument is passed (i.e., it has its default value), a *Session* object will instantiate and use a `FileSystemStore` store, keeping data in the `jobs` subdirectory of the session directory.

Methods *add* and *remove* are provided to manage the collection; the *len()* operator returns the number of tasks in the session; iteration over a session returns the tasks one by one:

```
>>> task1 = gc3libs.Task()
>>> id1 = session.add(task1)
>>> task2 = gc3libs.Task()
>>> id2 = session.add(task2)
>>> len(session)
2
>>> for t in session:
...     print(type(t))
<class 'gc3libs.Task'>
<class 'gc3libs.Task'>
>>> session.remove(id1)
>>> len(session)
1
```

When passed the *flush=False* optional argument, methods *add* and *remove* do not update the session metadata: i.e., the tasks are added or removed from the store and the in-memory task list, but the updated task list is not saved back to disk. This is useful when making many changes in a row; call *Session.flush* to persist the full set of changes.

The `Store` object is anyway accessible in the `store` attribute of each *Session* instance:

```
>>> type(session.store)
<class 'gc3libs.persistence.filesystem.FilesystemStore'>
```

However, *Session* defines methods *save* and *load* as a convenient proxy to the corresponding *Store* methods:

```
>>> obj = gc3libs.persistence.Persistable()
>>> oid = session.save(obj)
>>> obj2 = session.load(oid)
>>> obj.persistent_id == obj2.persistent_id
True
```

The whole session data can be removed by using method *destroy*:

```
>>> session.destroy()
>>> os.path.exists(session.path)
False
```

add(*task*, *flush=True*)
Add a *Task* to the current session, save it to the associated persistent storage, and return the assigned *persistent_id*:

```
>>> # create new, empty session
>>> import tempfile; tmpdir = tempfile.mktemp(dir='.')
>>> session = Session(tmpdir)
>>> len(session)
0

>>> # add a task to it
>>> task = gc3libs.Task()
>>> tid1 = session.add(task)
>>> len(session)
1
```

Duplicates are silently ignored: the same object can be added many times to the session, but gets the same ID each time:

```
>>> # add a different task
>>> tid2 = session.add(task)
>>> len(session)
1
>>> tid1 == tid2
True

>>> # do cleanup
>>> session.destroy()
>>> os.path.exists(session.path)
False
```

destroy()
Remove the session directory and all the tasks it contains from the store which are associated to this session.

Note: This will remove the associated task storage *if and only if* the storage is contained in the session directory!

---

flush()
Update session metadata.

Should be used after a save/remove operations, to ensure that the session state and metadata is correctly persisted.

forget(*task_id*, *flush=True*)
Remove task identified by *task_id* from the current session *but not* from the associated storage.

list_ids()
Return set of all task IDs belonging to this session.

list_names()
Return set of names of tasks belonging to this session.

load(*obj_id*)
Load an object from persistent storage and return it.

This is just a convenience proxy for calling method *load* on the *Store* instance associated with this session.

remove(*task_id*, *flush=True*)
Remove task identified by *task_id* from the current session *and* from the associated storage.

save(*obj*)
Save an object to the persistent storage and return *persistent_id* of the saved object.

This is just a convenience proxy for calling method *save* on the *Store* instance associated with this session.

The object is *not* added to the session, nor is session meta-data updated:

```
# create an empty session
>>> import tempfile; tmpdir = tempfile.mktemp(dir='.')
>>> session = Session(tmpdir)
>>> 0 == len(session)
True

# use `save` on an object
>>> obj = gc3libs.persistence.Persistable()
>>> oid = session.save(obj)

# session is still empty
>>> 0 == len(session)
True

# do cleanup
>>> session.destroy()
>>> os.path.exists(session.path)
False
```

save_all(*flush=True*)
Save all modified tasks to persistent storage.

set_end_timestamp(*time=None*)
Create a file named *finished* in the session directory. It's creation/modification time will be used to know when the session has finished.

Please note that *Session* does not know when a session is finished, so this method should be called by a `SessionBasedScript` class.

set_start_timestamp(*time=None*)
Create a file named *created* in the session directory. It's creation/modification time will be used to know when the session has sarted.

gc3libs.template

Support and expansion of programmatic templates.

The module *gc3libs.template* allows creation of textual templates with a simple object-oriented programming interface: given a string with a list of substitutions (using the syntax of Python's standard *substitute* module), a set of replacements can be specified, and the *gc3libs.template.expansions* function will generate all possible texts coming from the same template. Templates can be nested, and expansions generated recursviely.

class gc3libs.template.**Template**(*template*, *validator=<function <lambda>>*, ***extra_args*)
A template object is a pair *(obj, keywords)*. Methods are provided to substitute the keyword values into *obj*, and to iterate over expansions of the given keywords (optionally filtering the allowed combination of keyword values).

Second optional argument *validator* must be a function that accepts a set of keyword arguments, and returns *True* if the keyword combination is valid (can be expanded/substituted back into the template) or *False* if it should be discarded. The default validator passes any combination of keywords/values.

expansions(***keywords*)
Iterate over all valid expansions of the templated object *and* the template keywords. Returned items are *Template* instances constucted with the expanded template object and a valid combination of keyword values.

substitute(***extra_args*)

Return result of interpolating the value of keywords into the template. Keyword arguments *extra_args* can be used to override keyword values passed to the constructor.

If the templated object provides a *substitute* method, then return the result of invoking it with the template keywords as keyword arguments. Otherwise, return the result of applying Python standard library's *string.Template.safe_substitute()* on the string representation of the templated object.

Raise *ValueError* if the set of keywords/values is not valid according to the validator specified in the constructor.

gc3libs.template.**expansions**(*obj*, ***extra_args*)

Iterate over all expansions of a given object, recursively expanding all templates found. How the expansions are actually computed, depends on the type of object being passed in the first argument *obj*:

•If *obj* is a *list*, iterate over expansions of items in *obj*. (In particular, this flattens out nested lists.)

Example:

```
>>> L = [0, [2, 3]]
>>> list(expansions(L))
[0, 2, 3]
```

•If *obj* is a dictionary, return dictionary formed by all combinations of a key *k* in *obj* with an expansion of the corresponding value *obj[k]*. Expansions are computed by recursively calling *expansions(obj[k], **extra_args)*.

Example:

```
>>> D = {'a':1, 'b':[2,3]}
>>> E = list(expansions(D))
>>> len(E)
2
>>> {'a': 1, 'b': 2} in E
True
>>> {'a': 1, 'b': 3} in E
True
```

•If *obj* is a *tuple*, iterate over all tuples formed by the expansion of every item in *obj*. (Each item *t[i]* is expanded by calling *expansions(t[i], **extra_args)*.)

Example:

```
>>> T = (1, [2, 3])
>>> list(expansions(T))
[(1, 2), (1, 3)]
```

•If *obj* is a *Template* class instance, then the returned values are the result of applying the template to the expansion of each of its keywords.

Example:

```
>>> T1 = Template("a=${n}", n=[0,1])
>>> list(expansions(T1))
[Template('a=${n}', n=0), Template('a=${n}', n=1)]
```

Note that keywords passed to the *expand* invocation override the ones used in template construction:

```
>>> T2 = Template("a=${n}")
>>> list(expansions(T2, n=[1,3]))
[Template('a=${n}', n=1), Template('a=${n}', n=3)]

>>> T3 = Template("a=${n}", n=[0,1])
>>> list(expansions(T3, n=[2,3]))
[Template('a=${n}', n=2), Template('a=${n}', n=3)]
```

•Any other value is returned unchanged.

Example:

```
>>> V = 42
>>> list(expansions(V))
[42]
```

gc3libs.url

Utility classes and methods for dealing with URLs.

class `gc3libs.url.`**`Url`**

Represent a URL as a named-tuple object. This is an immutable object that cannot be changed after creation.

The following read-only attributes are defined on objects of class *Url*.

Attribute	Index	Value	if not present
scheme	0	URL scheme specifier	empty string
netloc	1	Network location part	empty string
path	2	Hierarchical path	empty string
query	3	Query component	empty string
hostname	4	Host name (lower case)	None
port	5	Port number as integer (if present)	None
username	6	User name	None
password	7	Password	None

There are two ways of constructing *Url* objects:

•By passing a string *urlstring*:

```
>>> u = Url('http://www.example.org/data')

>>> u.scheme
'http'
>>> u.netloc
'www.example.org'
>>> u.path
'/data'
```

The default URL scheme is `file`:

```
>>> u = Url('/tmp/foo')
>>> u.scheme
'file'
>>> u.path
'/tmp/foo'
```

Please note that extra leading slashes '/' are interpreted as the begining of a network location:

```
>>> u = Url('//foo/bar')
>>> u.path
'/bar'
>>> u.netloc
'foo'
>>> Url('///foo/bar').path
'/foo/bar'
```

Check RFC 3986 http://tools.ietf.org/html/rfc3986

If *force_abs* is *True* (default), then the *path* attribute is made absolute, by calling *os.path.abspath* if necessary:

```
>>> u = Url('foo/bar', force_abs=True)
>>> os.path.isabs(u.path)
True
```

Otherwise, if *force_abs* is *False*, then the *path* attribute stores the passed string unchanged:

```
>>> u = Url('foo', force_abs=False)
>>> os.path.isabs(u.path)
False
>>> u.path
'foo'
```

Other keyword arguments can specify defaults for missing parts of the URL:

```
>>> u = Url('/tmp/foo', scheme='file', netloc='localhost')
>>> u.scheme
'file'
>>> u.netloc
'localhost'
>>> u.path
'/tmp/foo'
```

- By passing keyword arguments only, to construct an *Url* object with exactly those values for the named fields:

```
>>> u = Url(scheme='http', netloc='www.example.org', path='/data')
```

In this form, the *force_abs* parameter is ignored.

See also: http://goo.gl/9WcRvR

adjoin(*relpath*)

Return a new *Url*, constructed by appending *relpath* to the path section of this URL.

Example:

```
>>> u0 = Url('http://www.example.org')
>>> u1 = u0.adjoin('data')
>>> str(u1)
'http://www.example.org/data'

>>> u2 = u1.adjoin('moredata')
>>> str(u2)
'http://www.example.org/data/moredata'
```

Even if *relpath* starts with */*, it is still appended to the path in the base URL:

```
>>> u3 = u2.adjoin('/evenmore')
>>> str(u3)
'http://www.example.org/data/moredata/evenmore'
```

class gc3libs.url.**UrlKeyDict**(*iter_or_dict=None*, *force_abs=False*)

A dictionary class enforcing that all keys are URLs.

Strings and/or objects returned by *urlparse* can be used as keys. Setting a string key automatically translates it to a URL:

```
>>> d = UrlKeyDict()
>>> d['/tmp/foo'] = 1
>>> for k in d.keys(): print (type(k), k.path)
(<class '....Url'>, '/tmp/foo')
```

Retrieving the value associated with a key works with both the string or the url value of the key:

```
>>> d['/tmp/foo']
1
```

```
>>> d[Url('/tmp/foo')]
1
```

Key lookup can use both the string or the *Url* value as well:

```
>>> '/tmp/foo' in d
True
>>> Url('/tmp/foo') in d
True
>>> 'file:///tmp/foo' in d
True
>>> 'http://example.org' in d
False
```

Class *UrlKeyDict* supports initialization by copying items from another *dict* instance or from an iterable of (key, value) pairs:

```
>>> d1 = UrlKeyDict({ '/tmp/foo':'foo', '/tmp/bar':'bar' })
>>> d2 = UrlKeyDict([ ('/tmp/foo', 'foo'), ('/tmp/bar', 'bar') ])
>>> d1 == d2
True
```

Differently from *dict*, initialization from keyword arguments alone is *not* supported:

```
>>> d3 = UrlKeyDict(foo='foo')
Traceback (most recent call last):
    ...
TypeError: __init__() got an unexpected keyword argument 'foo'
```

An empty *UrlKeyDict* instance is returned by the constructor when called with no parameters:

```
>>> d0 = UrlKeyDict()
>>> len(d0)
0
```

If *force_abs* is *True*, then all paths are converted to absolute ones in the dictionary keys.

```
>>> d = UrlKeyDict(force_abs=True)
>>> d['foo'] = 1
>>> for k in d.keys(): print os.path.isabs(k.path)
True
```

```
>>> d = UrlKeyDict(force_abs=False)
>>> d['foo'] = 2
>>> for k in d.keys(): print os.path.isabs(k.path)
False
```

class gc3libs.url.**UrlValueDict**(*iter_or_dict=None*, *force_abs=False*, ***extra_args*)

A dictionary class enforcing that all values are URLs.

Strings and/or objects returned by *urlparse* can be used as values. Setting a string value automatically translates it to a URL:

```
>>> d = UrlValueDict()
>>> d[1] = '/tmp/foo'
>>> d[2] = Url('file:///tmp/bar')
>>> for v in d.values(): print (type(v), v.path)
(<class '....Url'>, '/tmp/foo')
(<class '....Url'>, '/tmp/bar')
```

Retrieving the value associated with a key always returns the URL-type value, regardless of how it was set:

```
>>> repr(d[1]) == "Url(scheme='file', netloc='', path='/tmp/foo', "          "hostname=None, p
True
```

Class *UrlValueDict* supports initialization by any of the methods that work with a plain *dict* instance:

```
>>> d1 = UrlValueDict({ 'foo':'/tmp/foo', 'bar':'/tmp/bar' })
>>> d2 = UrlValueDict([ ('foo', '/tmp/foo'), ('bar', '/tmp/bar') ])
>>> d3 = UrlValueDict(foo='/tmp/foo', bar='/tmp/bar')

>>> d1 == d2
True
>>> d2 == d3
True
```

In particular, an empty *UrlDict* instance is returned by the constructor when called with no parameters:

```
>>> d0 = UrlValueDict()
>>> len(d0)
0
```

If *force_abs* is *True*, then all paths are converted to absolute ones in the dictionary values.

```
>>> d = UrlValueDict(force_abs=True)
>>> d[1] = 'foo'
>>> for v in d.values(): print os.path.isabs(v.path)
True
```

```
>>> d = UrlValueDict(force_abs=False)
>>> d[2] = 'foo'
>>> for v in d.values(): print os.path.isabs(v.path)
False
```

gc3libs.utils

Generic Python programming utility functions.

This module collects general utility functions, not specifically related to GC3Libs. A good rule of thumb for determining if a function or class belongs in here is the following: place a function or class in this module if you could copy its code into the sources of a different project and it would not stop working.

class gc3libs.utils.**Enum**

A generic enumeration class. Inspired by: http://goo.gl/1AL5N0 with some more syntactic sugar added.

An *Enum* class must be instanciated with a list of strings, that make the enumeration "label":

```
>>> Animal = Enum('CAT', 'DOG')
```

Each label is available as an instance attribute, evaluating to itself:

```
>>> Animal.DOG
'DOG'

>>> Animal.CAT == 'CAT'
True
```

As a consequence, you can test for presence of an enumeration label by string value:

```
>>> 'DOG' in Animal
True
```

Finally, enumeration labels can also be iterated upon:

```
>>> for a in sorted(Animal): print a
CAT
DOG
```

class gc3libs.utils.**ExponentialBackoff**(*slot_duration=0.05*, *max_retries=5*)

Generate waiting times with the exponential backoff algorithm.

Returned times are in seconds (or fractions thereof); they are integral multiples of the basic time slot, which is set with the *slot_duration* constructor parameter.

After *max_retries* have been attempted, any call to this iterator will raise a *StopIteration* exception.

The *ExponentialBackoff* class implements the iterator protocol, so you can just retrieve waiting times with the *.next()* method, or by looping over it:

```
>>> random.seed(314) # not-so-random for testing purposes...
>>> for wt in ExponentialBackoff():
...   print wt,
...
0.0 0.0 0.0 0.25 0.15 0.3
```

next()
Return next waiting time.

wait()
Wait for another while.

class gc3libs.utils.**History**
A list of messages with timestamps and (optional) tags.

The *append* method should be used to add a message to the *History*:

```
>>> L = History()
>>> L.append('first message')
>>> L.append('second one')
```

The *last* method returns the text of the last message appended, with its timestamp:

```
>>> L.last().startswith('second one at')
True
```

Iterating over a *History* instance returns message texts in the temporal order they were added to the list, with their timestamp:

```
>>> for msg in L: print(msg)
first message ...
```

append(*message*, **tags*)
Append a message to this *History*.

The message is timestamped with the time at the moment of the call.

The optional *tags* argument is a sequence of strings. Tags are recorded together with the message and may be used to filter log messages given a set of labels. *(This feature is not yet implemented.)*

format_message(*message*)
Return a formatted message, appending to the message its timestamp in human readable format.

last()
Return text of last message appended. If log is empty, return empty string.

class gc3libs.utils.**PlusInfinity**
An object that is greater-than any other object.

```
>>> x = PlusInfinity()
```

```
>>> x > 1
True
>>> 1 < x
True
>>> 12456324791025098345701248710234872359876345187 45 < x
True
```

```
>>> x > sys.maxint
True
>>> x < sys.maxint
False
>>> sys.maxint < x
True
```

PlusInfinity objects are actually larger than *any* given Python object:

```
>>> x > 'azz'
True
>>> x > object()
True
```

Note that *PlusInfinity* is a singleton, therefore you always get the same instance when calling the class constructor:

```
>>> x = PlusInfinity()
>>> y = PlusInfinity()
>>> x is y
True
```

Relational operators try to return the correct value when comparing *PlusInfinity* to itself:

```
>>> x < y
False
>>> x <= y
True
>>> x == y
True
>>> x >= y
True
>>> x > y
False
```

class gc3libs.utils.**Singleton**

Derived classes of *Singleton* can have only one instance in the running Python interpreter.

```
>>> x = Singleton()
>>> y = Singleton()
>>> x is y
True
```

class gc3libs.utils.**Struct**(*initializer=None*, ***extra_args*)

A *dict*-like object, whose keys can be accessed with the usual '[...]' lookup syntax, or with the '.' get attribute syntax.

Examples:

```
>>> a = Struct()
>>> a['x'] = 1
>>> a.x
1
>>> a.y = 2
>>> a['y']
2
```

Values can also be initially set by specifying them as keyword arguments to the constructor:

```
>>> a = Struct(z=3)
>>> a['z']
3
>>> a.z
3
```

Like *dict* instances, *Struct's have a 'copy* method to get a shallow copy of the instance:

```
>>> b = a.copy()
>>> b.z
3
```

copy()
Return a (shallow) copy of this *Struct* instance.

class gc3libs.utils.**YieldAtNext**(*generator*)
Provide an alternate protocol for generators.

Wrap a Python generator object, and buffer the return values from *send* and *throw* calls, returning *None* instead. Return the yielded value –or raise the *StopIteration* exception– upon the subsequent call to the *next* method.

gc3libs.utils.**backup**(*path*)
Rename the filesystem entry at *path* by appending a unique numerical suffix; return new name.

For example,

1.create a test file:

```
>>> import tempfile
>>> path = tempfile.mkstemp()[1]
```

2.then make a backup of it; the backup will end in `.~1~`:

```
>>> path1 = backup(path)
>>> os.path.exists(path + '.~1~')
True
```

3. re-create the file, and make a second backup: this time the file will be renamed with a `.~2~` extension:

```
>>> open(path, 'w').close()
>>> path2 = backup(path)
>>> os.path.exists(path + '.~2~')
True
```

cleaning up tests

```
>>> os.remove(path+'.~1~')
>>> os.remove(path+'.~2~')
```

gc3libs.utils.**basename_sans**(*path*)
Return base name without the extension.

gc3libs.utils.**cache_for**(*lapse*)
Cache the result of a (nullary) method invocation for a given amount of time. Use as a decorator on object methods whose results are to be cached.

Store the result of the first invocation of the decorated method; if another invocation happens before *lapse* seconds have passed, return the cached value instead of calling the real function again. If a new call happens after the grace period has expired, call the real function and store the result in the cache.

Note: Do not use with methods that take keyword arguments, as they will be discarded! In addition, arguments are compared to elements in the cache by *identity*, so that invoking the same method with equal but distinct object will result in two separate copies of the result being computed and stored in the cache.

Cache results and timestamps are stored into the objects' *_cache_value* and *_cache_last_updated* attributes, so the caches are destroyed with the object when it goes out of scope.

The working of the cached method can be demonstrated by the following simple code:

```
>>> class X(object):
...     def __init__(self):
...         self.times = 0
...     @cache_for(2)
```

```
    ...     def foo(self):
    ...             self.times += 1
    ...             return ("times effectively run: %d" % self.times)
    >>> x = X()
    >>> x.foo()
    'times effectively run: 1'
    >>> x.foo()
    'times effectively run: 1'
    >>> time.sleep(3)
    >>> x.foo()
    'times effectively run: 2'
```

gc3libs.utils.**cat**(**args*, ***extra_args*)
Concatenate the contents of all *args* into *output*. Both *output* and each of the *args* can be a file-like object or a string (indicating the path of a file to open).

If *append* is *True*, then *output* is opened in append-only mode; otherwise it is overwritten.

gc3libs.utils.**copy_recursively**(*src*, *dst*, *overwrite=False*, *changed_only=True*)
Copy *src* to *dst*, descending it recursively if necessary.

The *overwrite* and *changed_only* optional arguments have the same effect as in *copytree()* (which see).

gc3libs.utils.**copyfile**(*src*, *dst*, *overwrite=False*, *changed_only=True*, *link=False*)
Copy a file from *src* to *dst*; return *True* if the copy was actually made.

If *overwrite* is `False` (default), an existing destination entry is left unchanged and *False* is returned.

If *overwrite* is `True`, then *changed_only* determines if the destination file is overwritten:

- if *changed_only* is `True` (default), then destination is overwritten if and only if it has a different size or has been modified less recently than the source;
- if *changed_only* is `False`, then the destination is overwritten unconditionally.

If *link* is *True*, an attempt at hard-linking is done first; failing that, we copy the source file onto the destination one. Permission bits and modification times are copied as well.

If *dst* is a directory, a file with the same basename as *src* is created (or overwritten) in the directory specified.

Return `True` or `False`, depending on whether the source file was actually copied (or linked) to the destination.

gc3libs.utils.**copytree**(*src*, *dst*, *overwrite=False*, *changed_only=True*)
Recursively copy an entire directory tree rooted at *src*.

If *overwrite* is `False` (default), entries that already exist in the destination tree are left unchanged and not overwritten.

If *overwrite* is `True`, then *changed_only* determines which files are overwritten:

- if *changed_only* is `True` (default), then only files for which the source has a different size or has been modified more recently than the destination are copied;
- if *changed_only* is `False`, then *all* files in *source* will be copied into *destination*, unconditionally.

Destination directory *dst* is created if it does not exist.

See also: *shutil.copytree*.

gc3libs.utils.**count**(*seq*, *predicate*)
Return number of items in *seq* that match *predicate*. Argument *predicate* should be a callable that accepts one argument and returns a boolean.

gc3libs.utils.**defproperty**(*fn*)
Decorator to define properties with a simplified syntax in Python 2.4. See http://goo.gl/IoOZ8m for details and examples.

gc3libs.utils.**deploy_configuration_file**(*filename*, *template_filename=None*)
Ensure that configuration file *filename* exists; possibly copying it from the specified *template_filename*.

Return *True* if a file with the specified name exists in the configuration directory. If not, try to copy the template file over and then return *False*; in case the copy operations fails, a *NoConfigurationFile* exception is raised.

The *template_filename* is always resolved relative to GC3Libs' 'package resource' directory (i.e., the `etc/` directory in the sources. If *template_filename* is *None*, then it is assumed to be the base name of *filename*.

gc3libs.utils.**dirname**(*pathname*)
Same as *os.path.dirname* but return . in case of path names with no directory component.

gc3libs.utils.**fgrep**(*literal*, *filename*)
Iterate over all lines in a file that contain the *literal* string.

gc3libs.utils.**first**(*seq*)
Return the first element of sequence or iterator *seq*. Raise *TypeError* if the argument does not implement either of the two interfaces.

Examples:

```
>>> s = [0, 1, 2]
>>> first(s)
0

>>> s = {'a':1, 'b':2, 'c':3}
>>> first(sorted(s.keys()))
'a'
```

gc3libs.utils.**from_template**(*template*, ***extra_args*)
Return the contents of *template*, substituting all occurrences of Python formatting directives '*%*(key)s' with the corresponding values taken from dictionary *extra_args*.

If *template* is an object providing a *read()* method, that is used to gather the template contents; else, if a file named *template* exists, the template contents are read from it; otherwise, *template* is treated like a string providing the template contents itself.

gc3libs.utils.**getattr_nested**(*obj*, *name*)
Like Python's *getattr*, but perform a recursive lookup if *name* contains any dots.

gc3libs.utils.**grep**(*pattern*, *filename*)
Iterate over all lines in a file that match the *pattern* regular expression.

gc3libs.utils.**ifelse**(*test*, *if_true*, *if_false*)
Return *if_true* is argument *test* evaluates to *True*, return *if_false* otherwise.

This is just a workaround for Python 2.4 lack of the conditional assignment operator:

```
>>> a = 1
>>> b = ifelse(a, "yes", "no"); print b
yes
>>> b = ifelse(not a, 'yay', 'nope'); print b
nope
```

gc3libs.utils.**irange**(*start*, *stop*, *step=1*)
Iterate over all values greater or equal than *start* and less than *stop*. (Or the reverse, if *step* < *0*.)

Example:

```
>>> list(irange(1, 5))
[1, 2, 3, 4]
>>> list(irange(0, 8, 3))
[0, 3, 6]
>>> list(irange(8, 0, -2))
[8, 6, 4, 2]
```

Unlike the built-in *range* function, *irange* also accepts floating-point values:

```
>>> list(irange(0.0, 1.0, 0.5))
[0.0, 0.5]
```

Also unlike the built-in *range*, *both start* and *stop* have to be specified:

```
>>> irange(42)
Traceback (most recent call last):
  ...
TypeError: irange() takes at least 2 arguments (1 given)
```

Of course, a null *step* is not allowed:

```
>>> list(irange(1, 2, 0))
Traceback (most recent call last):
  ...
AssertionError: Null step in irange.
```

gc3libs.utils.**lock**(*path*, *timeout*, *create=True*)
Lock the file at *path*. Raise a *LockTimeout* error if the lock cannot be acquired within *timeout* seconds.

Return a *lock* object that should be passed unchanged to the *gc3libs.utils.unlock* function.

If no *path* points to a non-existent location, an empty file is created before attempting to lock (unless *create* is *False*). An attempt is made to remove the file in case an error happens.

See also: *gc3libs.utils.unlock()*

gc3libs.utils.**mkdir**(*path*, *mode=511*)
Like *os.makedirs*, but does not throw an exception if PATH already exists.

gc3libs.utils.**mkdir_with_backup**(*path*, *mode=511*)
Like *os.makedirs*, but if *path* already exists and is not empty, rename the existing one to a backup name (see the *backup* function).

Unlike *os.makedirs*, no exception is thrown if the directory already exists and is empty, but the target directory permissions are not altered to reflect *mode*.

gc3libs.utils.**move_recursively**(*src*, *dst*, *overwrite=False*, *changed_only=True*)
Move *src* to *dst*, descending it recursively if necessary.

The *overwrite* and *changed_only* optional arguments have the same effect as in *copytree()* (which see).

gc3libs.utils.**movefile**(*src*, *dst*, *overwrite=False*, *changed_only=True*, *link=False*)
Move a file from *src* to *dst*; return *True* if the move was actually made.

The *overwrite* and *changed_only* optional arguments have the same effect as in *copyfile()* (which see).

If *dst* is a directory, a file with the same basename as *src* is created (or overwritten) in the directory specified.

Return `True` or `False`, depending on whether the source file was actually moved to the destination.

See also: *copyfile()*

gc3libs.utils.**movetree**(*src*, *dst*, *overwrite=False*, *changed_only=True*)
Recursively move an entire directory tree rooted at *src*.

The *overwrite* and *changed_only* optional arguments have the same effect as in *copytree()* (which see).

See also: *copytree()*.

gc3libs.utils.**occurs**(*pattern*, *filename*)
Return `True` if any line in *filename* matches regular expression *pattern*.

gc3libs.utils.**prettyprint**(*D*, *indent=0*, *width=0*, *maxdepth=None*, *step=4*, *only_keys=None*, *output=<open file '<stdout>', mode 'w'>*, *_key_prefix=''*, *_exclude=None*)
Print dictionary instance *D* in a YAML-like format. Each output line consists of:

- *indent* spaces,
- the key name,
- a colon character `:`,
- the associated value.

If the total line length exceeds *width*, the value is printed on the next line, indented by further *step* spaces; a value of 0 for *width* disables this line wrapping.

Optional argument *only_keys* can be a callable that must return *True* when called with keys that should be printed, or a list of key names to print.

Dictionary instances appearing as values are processed recursively (up to *maxdepth* nesting). Each nested instance is printed indented *step* spaces from the enclosing dictionary.

`gc3libs.utils.`**`progressive_number`**(*qty=None*, *id_filename=None*)

Return a positive integer, whose value is guaranteed to be monotonically increasing across different invocations of this function, and also across separate instances of the calling program.

This is accomplished by using a system-wide file which holds the "next available" ID. The location of this file can be set using the `GC3PIE_ID_FILE` environment variable, or programmatically using the *id_filename* argument. By default, the "next ID" file is located at `~/.gc3/next_id.txt`:file:

Example:

```
>>> # create "next ID" file in a temporary location
>>> import tempfile, os
>>> (fd, tmp) = tempfile.mkstemp()

>>> n = progressive_number(id_filename=tmp)
>>> m = progressive_number(id_filename=tmp)
>>> m > n
True
```

If you specify a positive integer as argument, then a list of monotonically increasing numbers is returned. For example:

```
>>> ls = progressive_number(5, id_filename=tmp)
>>> len(ls)
5
```

(clean up test environment)

```
>>> os.remove(tmp)
```

In other words, *progressive_number(N)* is equivalent to:

```
nums = [ progressive_number() for n in range(N) ]
```

only more efficient, because it has to obtain and release the lock only once.

After every invocation of this function, the last returned number is stored into the file passed as argument *id_filename*. If the file does not exist, an attempt to create it is made before allocating an id; the method can raise an *IOError* or *OSError* if *id_filename* cannot be opened for writing.

Note: as file-level locking is used to serialize access to the counter file, this function may block (default timeout: 30 seconds) while trying to acquire the lock, or raise a *LockTimeout* exception if this fails.

Raise LockTimeout, IOError, OSError

Returns A positive integer number, monotonically increasing with every call. A list of such numbers if argument *qty* is a positive integer.

`gc3libs.utils.`**`read_contents`**(*path*)

Return the whole contents of the file at *path* as a single string.

Example:

```
>>> read_contents('/dev/null')
''


>>> import tempfile
>>> (fd, tmpfile) = tempfile.mkstemp()
>>> w = open(tmpfile, 'w')
>>> w.write('hey')
>>> w.close()
>>> read_contents(tmpfile)
'hey'
```

(If you run this test, remember to do cleanup afterwards)

```
>>> os.remove(tmpfile)
```

gc3libs.utils.**safe_repr**(*obj*)
Return a string describing Python object *obj*.

Avoids calling any Python magic methods, so should be safe to use as a 'last resort' in implementation of *__str__* and *__repr__*.

gc3libs.utils.**same_docstring_as**(*referenced_fn*)
Function decorator: sets the docstring of the following function to the one of *referenced_fn*.

Intended usage is for setting docstrings on methods redefined in derived classes, so that they inherit the docstring from the corresponding abstract method in the base class.

gc3libs.utils.**samefile**(*path1*, *path2*)
Like *os.path.samefile* but return *False* if either one of the paths does not exist.

gc3libs.utils.**sh_quote_safe**(*text*)
Escape a string for safely passing as argument to a shell command.

Return a single-quoted string that expands to the exact literal contents of *text* when used as an argument to a shell command. Examples (note that backslashes are doubled because of Python's string read syntax):

```
>>> print(sh_quote_safe("arg"))
'arg'
>>> print(sh_quote_safe("'arg'"))
''\''arg'\'''
```

gc3libs.utils.**sh_quote_safe_cmdline**(*args*)
Single-quote a list of strings for passing to the shell as a command. Return the list of quoted arguments concatenated and separated by spaces.

Examples:

```
>>> sh_quote_safe_cmdline(['sh', '-c', 'echo c(1,2,3)'])
"'sh' '-c' 'echo c(1,2,3)'"
```

gc3libs.utils.**sh_quote_unsafe**(*text*)
Double-quote a string for passing as argument to a shell command.

Return a double-quoted string that expands to the contents of *text* but still allows variable expansion and \-escapes processing by the UNIX shell. Examples (note that backslashes are doubled because of Python's string read syntax):

```
>>> print(sh_quote_unsafe("arg"))
"arg"
>>> print(sh_quote_unsafe('"arg"'))
"\"arg\""
>>> print(sh_quote_unsafe(r'"\"arg\""'))
"\"\\\"arg\\\"\""
```

gc3libs.utils.**sh_quote_unsafe_cmdline**(*args*)

Double-quote a list of strings for passing to the shell as a command. Return the list of quoted arguments concatenated and separated by spaces.

Examples:

```
>>> sh_quote_unsafe_cmdline(['sh', '-c', 'echo $HOME'])
'"sh" "-c" "echo $HOME"'
```

gc3libs.utils.**string_to_boolean**(*word*)

Convert *word* to a Python boolean value and return it. The strings *true*, *yes*, *on*, *1* (with any capitalization and any amount of leading and trailing spaces) are recognized as meaning Python *True*:

```
>>> string_to_boolean('yes')
True
>>> string_to_boolean('Yes')
True
>>> string_to_boolean('YES')
True
>>> string_to_boolean(' 1 ')
True
>>> string_to_boolean('True')
True
>>> string_to_boolean('on')
True
```

Any other word is considered as boolean *False*:

```
>>> string_to_boolean('no')
False
>>> string_to_boolean('No')
False
>>> string_to_boolean('Nay!')
False
>>> string_to_boolean('woo-hoo')
False
```

This includes also the empty string and whitespace-only:

```
>>> string_to_boolean('')
False
>>> string_to_boolean('  ')
False
```

gc3libs.utils.**stripped**(*iterable*)

Iterate over lines in *iterable* and return each of them stripped of leading and trailing blanks.

gc3libs.utils.**tempdir**(**args*, ***kwds*)

A context manager for creating and then deleting a temporary directory.

All arguments are passed unchanged to the *tempfile.mkdtemp* standand library function.

(Original source and credits: http://stackoverflow.com/a/10965572/459543)

gc3libs.utils.**test_file**(*path*, *mode*, *exception=<type 'exceptions.RuntimeError'>*, *isdir=False*)

Test for access to a path; if access is not granted, raise an instance of *exception* with an appropriate error message. This is a frontend to `os.access()`, which see for exact semantics and the meaning of *path* and *mode*.

Parameters

- **path** – Filesystem path to test.
- **mode** – See `os.access()`
- **exception** – Class of exception to raise if test fails.

- **isdir** – If *True* then also test that *path* points to a directory.

If the test succeeds, *True* is returned:

```
>>> test_file('/bin/cat', os.F_OK)
True
>>> test_file('/bin/cat', os.R_OK)
True
>>> test_file('/bin/cat', os.X_OK)
True
>>> test_file('/tmp', os.X_OK)
True
```

However, if the test fails, then an exception is raised:

```
>>> test_file('/bin/cat', os.W_OK)
Traceback (most recent call last):
  ...
RuntimeError: Cannot write to file '/bin/cat'.
```

If the optional argument *isdir* is *True*, then additionally test that *path* points to a directory inode:

```
>>> test_file('/tmp', os.F_OK, isdir=True)
True

>>> test_file('/bin/cat', os.F_OK, isdir=True)
Traceback (most recent call last):
  ...
RuntimeError: Expected '/bin/cat' to be a directory, but it's not.
```

gc3libs.utils.**to_bytes**(*s*)

Convert string *s* to an integer number of bytes. Suffixes like 'KB', 'MB', 'GB' (up to 'YB'), with or without the trailing 'B', are allowed and properly accounted for. Case is ignored in suffixes.

Examples:

```
>>> to_bytes('12')
12
>>> to_bytes('12B')
12
>>> to_bytes('12KB')
12000
>>> to_bytes('1G')
1000000000
```

Binary units 'KiB', 'MiB' etc. are also accepted:

```
>>> to_bytes('1KiB')
1024
>>> to_bytes('1MiB')
1048576
```

gc3libs.utils.**touch**(*path*)

Ensure a regular file exists at *path*.

If the file already exists, its access and modification time are updated.

(This is a very limited and stripped down version of the `touch` POSIX utility.)

gc3libs.utils.**uniq**(*seq*)

Iterate over all unique elements in sequence *seq*.

Distinct values are returned in a sorted fashion.

Examples:

```
>>> for value in uniq([4,1,1,2,3,1,2]): print value
...
1
2
3
4
```

```
>>> for value in uniq([1,2,3,4]): print value
...
1
2
3
4
```

```
>>> for value in uniq([1,1,1,1]): print value
...
1
```

gc3libs.utils.**unlock**(*lock*)
Release a previously-acquired lock.

Argument *lock* should be the return value of a previous *gc3libs.utils.lock* call.

See also: *gc3libs.utils.lock()*

gc3libs.utils.**update_parameter_in_file**(*path*, *var_in*, *new_val*, *regex_in*)
Updates a parameter value in a parameter file using predefined regular expressions in *_loop_regexps*.

Parameters

- **path** – Full path to the parameter file.
- **var_in** – The variable to modify.
- **new_val** – The updated parameter value.
- **regex** – Name of the regular expression that describes the format of the parameter file.

gc3libs.utils.**write_contents**(*path*, *data*)
Overwrite the contents of the file at *path* with the given data. If the file does not exist, it is created.

Example:

```
>>> import tempfile
>>> (fd, tmpfile) = tempfile.mkstemp()
>>> write_contents(tmpfile, 'big data here')
>>> read_contents(tmpfile)
'big data here'
```

(If you run this test, remember to clean up afterwards)

```
>>> os.remove(tmpfile)
```

gc3libs.workflow

Implementation of task collections.

Tasks can be grouped into collections, which are tasks themselves, therefore can be controlled (started/stopped/cancelled) like a single whole. Collection classes provided in this module implement the basic patterns of job group execution; they can be combined to form more complex workflows. Hook methods are provided so that derived classes can implement problem-specific job control policies.

class gc3libs.workflow.**AbortOnError**
Mix-in class to make a *SequentialTaskCollection* turn to TERMINATED state as soon as one of the tasks fail.

A second effect of mixing this class in is that the *self.execution.returncode* mirrors the return code of the last finished task.

class gc3libs.workflow.**DependentTaskCollection**(*tasks=None*, ***extra_args*)
Run a set of tasks, respecting inter-dependencies between them.

Each task can list a number of tasks that need to be run before it; upon submission, a *DependentTaskCollection* creates a direct acyclic graph from that dependency information and ensures that no task is run before its dependencies have been successfully executed.

The collection state is set to *TERMINATED* once all tasks have reached the same terminal status.

add(*task*, *after=None*)
Add a task to the collection.

The task will be run after any tasks referenced in the *after* sequence have terminated their run. Alternatively, a task can list tasks it depends upon in its `.after` attribute; i.e., the following two syntaxes are equivalent:

```
>>> coll.add(task1, after=[task2])
```

```
>>> task1.after = [task2]
>>> coll.add(task1)
```

Note: tasks can only be added to a *DependentTaskCollection* while it's in state `NEW`.

class gc3libs.workflow.**ParallelTaskCollection**(*tasks=None*, ***extra_args*)
A *ParallelTaskCollection* runs all of its tasks concurrently.

The collection state is set to *TERMINATED* once all tasks have reached the same terminal status.

add(*task*)
Add a task to the collection.

attach(*controller*)
Use the given Controller interface for operations on the job associated with this task.

kill(***extra_args*)
Terminate all tasks in the collection, and set collection state to *TERMINATED*.

progress()
Try to advance all jobs in the collection to the next state in a normal lifecycle. Return list of task execution states.

submit(*resubmit=False*, *targets=None*, ***extra_args*)
Start all tasks in the collection.

update_state(***extra_args*)
Update state of all tasks in the collection.

class gc3libs.workflow.**RetryableTask**(*task*, *max_retries=0*, ***extra_args*)
Wrap a *Task* instance and re-submit it until a specified termination condition is met.

By default, the re-submission upon failure happens iff execution terminated with nonzero return code; the failed task is retried up to *self.max_retries* times (indefinitely if *self.max_retries* is 0).

Override the *retry* method to implement a different retryal policy.

Note: The resubmission code is implemented in the `terminated()`, so be sure to call it if you override in derived classes.

changed
Evaluates to *True* if this task or any of its subtasks has been modified and should be saved to persistent storage.

retry()
Return *True* or *False*, depending on whether the failed task should be re-submitted or not.

The default behavior is to retry a task iff its execution terminated with nonzero returncode and the maximum retry limit has not been reached. If *self.max_retries* is 0, then the dependent task is retried indefinitely.

Override this method in subclasses to implement a different policy.

update_state()
Update the state of the dependent task, then resubmit it if it's TERMINATED and *self.retry()* is *True*.

class gc3libs.workflow.**SequentialTaskCollection**(*tasks*, ***extra_args*)
A *SequentialTaskCollection* runs its tasks one at a time.

After a task has completed, the *next* method is called with the index of the finished task in the *self.tasks* list; the return value of the *next* method is then made the collection *execution.state*. If the returned state is *RUNNING*, then the subsequent task is started, otherwise no action is performed.

The default *next* implementation just runs the tasks in the order they were given to the constructor, and sets the state to *TERMINATED* when all tasks have been run.

attach(*controller*)
Use the given Controller interface for operations on the job associated with this task.

kill(***extra_args*)
Stop execution of this sequence. Kill currently-running task (if any), then set collection state to TERMINATED.

next(*done*)
Return the state or task to run when step number *done* is completed.

This method is called when a task is finished; the *done* argument contains the index number of the just-finished task into the *self.tasks* list. In other words, the task that just completed is available as *self.tasks[done]*.

The return value from *next* can be either a task state (i.e., an instance of *Run.State*), or a valid index number for *self.tasks*. In the first case:

- if the return value is *Run.State.TERMINATED*, then no other jobs will be run;
- otherwise, the return value is assigned to *execution.state* and the next job in the *self.tasks* list is executed.

If instead the return value is a (nonnegative) number, then tasks in the sequence will be re-run starting from that index.

The default implementation runs tasks in the order they were given to the constructor, and sets the state to TERMINATED when all tasks have been run. This method can (and should) be overridden in derived classes to implement policies for serial job execution.

submit(*resubmit=False*, *targets=None*, ***extra_args*)
Start the current task in the collection.

update_state(***extra_args*)
Update state of the collection, based on the jobs' statuses.

class gc3libs.workflow.**StagedTaskCollection**(***extra_args*)
Simplified interface for creating a sequence of Tasks. This can be used when the number of Tasks to run is fixed and known at program writing time.

A *StagedTaskCollection* subclass should define methods *stage0*, *stage1*, ... up to *stageN* (for some arbitrary value of N positive integer). Each of these *stageN* must return a `Task` instance; the task returned by the *stage0* method will be executed first, followed by the task returned by *stage1*, and so on. The sequence stops at the first N such that *stageN* is not defined.

The exit status of the whole sequence is the exit status of the last *Task* instance run. However, if any of the *stageX* methods returns an integer value instead of a *Task* instance, then the sequence stops and that number is used as the sequence exit code.

class gc3libs.workflow.**StopOnError**
Mix-in class to make a *SequentialTaskCollection* turn to STOPPED state as soon as one of the tasks fail.

A second effect of mixing this class in is that the *self.execution.returncode* mirrors the return code of the last finished task.

class gc3libs.workflow.**TaskCollection**(*tasks=None*, ***extra_args*)
Base class for all task collections. A "task collection" is a group of tasks, that can be managed collectively as a single one.

A task collection implements the same interface as the *Task* class, so you can use a *TaskCollection* everywhere a *Task* is required. A task collection has a *state* attribute, which is an instance of *gc3libs.Run.State*; each concrete collection class decides how to deduce a collective state based on the individual task states.

add(*task*)
Add a task to the collection.

attach(*controller*)
Use the given Controller interface for operations on the job associated with this task.

changed
Evaluates to *True* if this task or any of its subtasks has been modified and should be saved to persistent storage.

free()
This method just asks the Engine to free the contained tasks.

iter_tasks()
Iterate over non-collection tasks enclosed in this collection.

iter_workflow()
Returns an iterator that will traverse the whole tree of tasks.

peek(*what*, *offset=0*, *size=None*, ***extra_args*)
Raise a *gc3libs.exceptions.InvalidOperation* error, as there is no meaningful semantics that can be defined for *peek* into a generic collection of tasks.

remove(*task*)
Remove a task from the collection.

stats(*only=None*)
Return a dictionary mapping each state name into the count of tasks in that state. In addition, the following keys are defined:

- *ok*: count of TERMINATED tasks with return code 0
- *failed*: count of TERMINATED tasks with nonzero return code
- *total*: count of managed tasks, whatever their state

If the optional argument *only* is not None, tasks whose class is not contained in *only* are ignored.

Parameters **only** (*tuple*) – Restrict counting to tasks of these classes.

terminated()
Called when the job state transitions to *TERMINATED*, i.e., the job has finished execution (with whatever exit status, see *returncode*) and the final output has been retrieved.

Default implementation for *TaskCollection* is to set the exitcode to the maximum of the exit codes of its tasks. If no tasks were run, the exitcode is set to 0.

update_state(***extra_args*)
Update the running state of all managed tasks.

gc3utils

gc3utils.commands

gc3utils.frontend

This is the main entry point for command **gc3utils** – a simple command-line frontend to distributed resources

This is a generic front-end code; actual implementation of commands can be found in `gc3utils.commands`

`gc3utils.frontend.`**`main`**`()`
Generic front-end function to invoke the commands in *gc3utils/commands.py*

2.3 Developer Documentation

This section contains all information needed for people who wants to contribute to GC3Pie.

2.3.1 Contributing to GC3Pie

First of all, thanks for wanting to contribute to GC3Pie! GC3Pie is an open-ended endeavour, and we're always looking for new ideas, suggestions, and new code. (And also, for fixes to bugs old and new ;-))

The paragraphs below should brief you about the organization of the GC3Pie code repositories, and the suggested guidelines for code and documentation style. Feel free to request more info or discuss the existing recommendations on the GC3Pie mailing list

Code repository organization

GC3Pie code is hosted in a Google Code repository, which you can access online or using any Subversion client. Refer to the checkout instructions to grab a copy of the sources.

Please note that anyone can read the sources, but you need to be granted *committer* status before you can make any modifications into the code; read section *how can I get access to the SVN repository?* below to request write-access to the repository.

Repository structure

The GC3Pie code repository follows the standard subversion layout:

- `trunk` is the place for development code: it has all the latest and greatest features, and also the newest and nastiest bugs.
- `tags` is where released code is: each subdirectory of `tags` is a snapshot of a release of GC3Pie code, and should never change.
- `branches` are alternative development lines, for instance code from past releases that still gets bugfixes, or experimental features that have not been implemented in the main development line `trunk` (because, e.g., they require a radical API change).

We shall now describe the contents of the `trunk` directory, as there is where most new code will land. Organization of the code in `tags` and `branches` is very similar and you should be able to adapt easily.

The `gc3pie` directory in `trunk` contains all GC3Pie code. It has one subdirectory for each of the main parts of GC3Pie:

- The `gc3libs` directory contains the GC3Libs code, which is the core of GC3Pie. GC3Libs are extensively described in the *API* section of this document; read the module descriptions to find out where your new suggested functionality would suit best. If unsure, ask on the GC3Pie mailing list.

- The `gc3utils` directory contains the sources for the low-level GC3Utils command-line utilities.
- The `gc3apps` directory contains the sources for higher level scripts that implement some computational use case of independent interest.

  The `gc3apps` directory contains one subdirectory per *application script*. Actually, each subdirectory can contain one or more Python scripts, as long as they form a coherent bundle; for instance, Rosetta is a suite of applications in computational biology: there are different GC3Apps script corresponding to different uses of the Rosetta suite, all of them grouped into the `rosetta` subdirectory.

  Subdirectories of the `gc3apps` directory follow this naming convention:

  – the directory name is the main application name, if the application that the scripts wrap is a known, publicly-released computational application (e.g., Rosetta, GAMESS)

  – the directory name is the requestor's name, if the application that the scripts wrap is some research code that is being internally developed. For instance, the `bf.uzh.ch` directory contains scripts that wrap code for economic simulations that is being developed at the Banking and Finance Institute of the University of Zurich

How can I get access to the SVN repository?

Please send an email to <gc3pie-dev@googlegroups.com>. Note that, in order to access the GC3Pie source repository you will need a Google Account, so sending the request email from a Gmail address might be a good idea.

Package generation

Due to issue 329, we don't use the automatic discovery *feature* of `setuptools`, so the files included in the distributed packages are those in the `MANIFEST.in` file, please check The MANIFEST.in template section of the python documentation for a syntax reference. We usually include only code, documentation, and related files. We also include the regression tests, but we **do not include** the application tests in `gc3apps/*/test` directories.

Testing the code

In developing GC3Pie we try to use a Test Driven Development approach, in the light of the quote: *It's tested or it's broken*. We use tox and nose as test runners, which make creating tests very easy.

Running the tests

You can both run tests on your current environment using *nosetests* or use *tox_* to create and run tests on separate environments. We suggest you to use *nosetests* while you are still fixing the problem, in order to be able to run only the failing test, but we strongly suggest you to run *tox before* committing your code.

Running tests with *nosetests* In order to have the *nosetests* program, you need to install *nose_* in your current environment **and** gc3pie must be installed in develop mode:

```
pip install nose
python setup.py develop
```

Then, from the top level directory, run the tests with:

```
nose -c nose.cfg
```

Nose will then crawl the directory tree looking for available tests. You can also specify a subset of the available sets, by:

- specifying the directory from which nose should start looking for tests:

```
# Run only backend-related tests
nose -c nose.cfg gc3libs/backends
```

- specifying the file containing the tests you want to run:

```
# Run only tests contained in a specific file
nose -c nose.cfg gc3libs/tests/test_session.py
```

- specifying the id of the test (you need to run nose at least one to know which id is assigned to each test):

```
# Run only test number 123
nose -c nose.cfg 123
```

Running multiple tests In order to test GC3Pie against multiple version of python we use tox, which creates virtual environments for all configured python version, runs nose inside each one of them, and prints a summary of the test results.

You don't need to have *tox* installed in the virtual environment you use to develop gc3pie, you can create a new virtual environment and install *tox* on it with:

```
pip install tox
```

Running tox is straightforward; just type `tox` on the command-line in GC3Pie's top level source directory.

The default `tox.ini` file shipped with GC3Pie attempts to test all Python versions from 2.4 to 2.7 (inclusive). If you want to run tests only for a specific version of python, for instance Python 2.6, use the `-e` option:

```
tox -e py26
[...]
Ran 118 tests in 14.168s

OK (SKIP=9)
_____________________________________________________ [tox summary] _____________________________________________________
[TOX] py26: commands succeeded
[TOX] congratulations :)
```

(See section *skipping tests* for a discussion about how and when to define skipped tests.)

Option `-r` instructs **`tox`** to re-build the testing virtual environment. This is usually needed when you update the dependencies of GC3Pie or when you add or remove command line programs or configuration files. However, if you feel that the environments can be *unclean*, you can clean up everything by:

1. deleting all the `*.pyc` file in your source tree:

```
find . -name '*.pyc' -delete
```

2. deleting and recreating tox virtual environments:

```
tox -r
```

Organizing tests

Each single python file should have a test file inside a `tests` subpackage with filename created by prefixing `test_` to the filename to test. For example, if you created a file `foo.py`, there should be a file `tests/test_foo.py` which will contains tests for `foo.py`.

Even though following the naming convention above is not always possible, each test regarding a specific component should be in a file inside a `tests` directory inside that component. For instance, tests for the subpackage *gc3libs.persistence* are located inside the directory `gc3libs/persistence/tests` but are not named after the specific file.

Writing tests

Please remember that it may be hard to understand, whenever a test fails, if it's a bug in the code or in the tests! Therefore please remember:

- Try to keep tests as simple as possible, and *always* simpler than the tested code. (*Debugging is twice as hard as writing the code in the first place.*, Brian W. Kernighan and P. J. Plauger)
- Write multiple indipendent tests to test different possible behavior and/or different methods of a class.
- Tests should cover methods and functions, but also specific use cases.
- If you are fixing a bug, it's good practice to write a test to check if the bug is still there, in order to avoid to re-include the bug in the future.
- Tests should clean up every temporary file they create.

Writing tests is very easy: just create a file whose name begins with `test_`, then put in it some functions which name begins with `test_`; the nose framework will automatically call each one of them. Moreover, nose will run also any doctest which will be found in the code.

Full documentation of the nose framework is available at the nose website. However, there are some of the interesting features you may want to use to improve your tests, detailed in the following sections.

Testing for errors If your test must verify that the code raises an exception, instead of wrapping the test inside a `try: ... except:` block you can use the *@raises* decorator from the *nose.tools* module:

```
from nose.tools import raises

@raises(TypeError)
def test_invalid_invocation():
    Application()
```

This is exactly the same as writing:

```
try:
    Application()
    assert False, "we should have got an exception"
except TypeError:
    pass
```

Skipping tests If you want to skip a test, just raise a *SkipTest* exception (imported from the *nose.plugins.skip* module). This is useful when you know that the test will fail, either because the code is not ready yet, or because some environmental conditions are not satisfied (e.g., an optional module is missing, or the code needs to access a service that is not available). For example:

```
from nose.plugins.skip import SkipTest
try:
    import MySQLdb
except ImportError:
    raise SkipTest("Error importing MySQL backend. Skipping MySQL low level tests")
```

Generating tests It is possible to use Python generators to create multiple tests at run time:

```
def test_evens():
    for i in range(0, 5):
        yield check_even, i, i*3

def check_even(n, nn):
    assert n % 2 == 0 or nn % 2 == 0
```

This will result in five tests: nose will iterate the generator, creating a function test case wrapper for each tuple it yields. Specifically, in the example above, nose will execute the function calls `check_even(0,0)`, `check_even(1,3)`, ..., `check_even(4,12)` as if each of them were written in the source as a separate test; if any of them fails (i.e., raises an *AssertionError*), then the test is considered failed.

Grouping tests into classes Tests that share the same set-up or clean-up code should be grouped into *test classes*:

- The exact same set-up and clean-up code *(fixtures)* will be run before and after each test, but is written down only once.
- Python class inheritance can be used to run the same tests on different configurations (e.g., by just overriding the set-up and clean-up code).

A test class is a regular Python class, whose name begins with `Test` (first letter must be uppercase); each method whose name begins with `test_` defines a test case.

If the class defines a *setUp* method, it will be called *before each test method*. If the class defines a *tearDown* method, it will be called *after each test method*.

If class methods `setup_class` and `teardown_class` are defined, nose will invoke them *once* (before and after performing the tests of that class, respectively).

A canonical example of a test class with fixtures looks like this:

```
class TestClass(object):

   @classmethod
   def setup_class(cls):
      ...

   @classmethod
   def teardown_class(cls):
      ...

   def setUp(self):
      ...

   def tearDown(self):
      ...

   def test_case_1(self):
      ...

   def test_case_2(self):
      ...

   def test_case_3(self):
      ...
```

The nose framework will execute a code like this:

```
TestClass.setup_class()
for test_method in get_test_classes():
   obj = TestClass()
   obj.setUp()
   try:
      obj.test_method()
   finally:
      obj.tearDown()
TestClass.teardown_class()
```

That is, for each test case, a new instance of the *TestClass* is created, set up, and torn down – thus approximating the Platonic ideal of running each test in a completely new, pristine environment.

Opening the python debugger while running a test

When running using *nosetests*:command you cannot just execute *pdb.set_trace()* to open a debugger console. However, you can run the *set_trace()* function of the *nose.tools* module:

```
import nose.tools; nose.tools.set_trace()
```

Coding style

Python code should be written according to PEP 8 recommendations. (And by this we mean not just the code style.)

Please take the time to read PEP 8 through, as it is widely-used across the Python programming community – it will benefit your contribution to any free/open-source Python project!

Anyway, here's a short summary for the impatient:

- use English nouns to name variables and classes; use verbs to name object methods.
- use 4 spaces to indent code; never use TABs.
- use lowercase letters for method and variable names; use underscores _ to separate words in multi-word identifiers (e.g., `lower_case_with_underscores`)
- use "CamelCase" for class and exception names.
- but, above all, do not blindly follow the rules and try to do the thing that *enhances code clarity and readability!*

Here's other code conventions that apply to GC3Pie code; since they are not always widely followed or known, a short rationale is given for each of them.

- Every class and function should have a docstring. Use reStructuredText markup for docstrings and documentation text files.

  Rationale: A concise English description of the purpose of a function can be faster to read than the code. Also, undocumented functions and classes do not appear in this documentation, which makes them invisible to new users.

- Use fully-qualified names for all imported symbols; i.e., write `import foo` and then use `foo.bar()` instead of `from foo import bar`. If there are few imports from a module, and the imported names do *clearly* belong to another module, this rule can be relaxed if this enhances readability, but *never* do use unqualified names for exceptions.

  Rationale: There are so many functions and classes in GC3Pie, so it may be hard to know to which module the function *count* belongs. (Think especially of people who have to bugfix a module they didn't write in the first place.)

- When calling methods or functions that accept both positional and optional arguments like:

```
def foo(a, b, key1=defvalue1, key2=defvalue2):
```

  always specify the argument name for optional arguments, which means **do not call**:

```
foo(1, 2, value1, value2)
```

  but **call instead**:

```
foo(1, 2, key1=value1, key2=value2)
```

  Rationale: calling the function with explicit argument names will reduce the risk of hit some compatibility issues. It is perfectly fine, from the point of view of the developer, to change the signature of a function by swapping two different *optional* arguments, so this change can happen any time, although changing *positional* arguments will break backward compatibility, and thus it's usually well advertised and tested.

- Use double quotes `"` to enclose strings representing messages meant for human consumption (e.g., log messages, or strings that will be printed on the users' terminal screen).

  Rationale: The apostrophe character `'` is a normal occurrence in English text; use of the double quotes minimizes the chances that you introduce a syntax error by terminating a string in its middle.

- Follow normal typographic conventions when writing user messages and output; prefer clarity and avoid ambiguity, even if this makes the messages longer.

  Rationale: Messages meant to be read by users *will* be read by users; and if they are not read by users, they will be fired back verbatim on the mailing list on the next request for support. So they'd better be clear, or you'll find yourself wondering what that message was intended to mean 6 months ago.

  Common typographical conventions enhance readability, and help users identify lines of readable text.

- Use single quotes `'` for strings that are meant for internal program usage (e.g., attribute names).

  Rationale: To distinguish them visually from messages to the user.

- Use triple quotes `"""` for docstrings, even if they fit on a single line.

  Rationale: Visual distinction.

- Each file should have this structure:
  - the first line is the hash-bang line,
  - the module docstring (explain briefly the module purpose and features),
  - the copyright and licence notice,
  - module imports (in the order suggested by **PEP 8**)
  - and then the code...

  Rationale: The docstring should be on top so it's the first thing one reads when inspecting a file. The copyright notice is just a waste of space, but we're required by law to have it.

Documentation

The documentation can be found in gc3pie/docs. It is generated using Sphinx (http://sphinx-doc.org/contents.html).

GC3Pie documentation is divided in three sections:

- *User Documentation*: info on how to install, configure and run GC3Pie applications.
- *Programmer Documentation*: info for programmers who want to use the GC3Pie libraries to write their own scripts and applications.
- *Developer Documentation*: detailed information on how to contribute to GC3Pie and get your code included in the main library.

The GC3Libs programming API is the most relevant part of the docs for developers contributing code and is generated automatically from the docstrings inside the modules. Automatic documentation in Sphinx is described under http://sphinx-doc.org/tutorial.html#autodoc. While updating the docs of existing modules is simply done by running `make html`, adding documentation for a new module requires one of the following two procedures:

- Add a reference to the new module in `docs/programmers/api/index.txt`. Additionally, create a file that enables automatic documentation for the module. For the module `core.py`, for example, automatic documentation is enabled by a file `docs/programmers/api/gc3libs/core.txt` with the following content:

```
`gc3libs.core`
==============

.. automodule:: gc3libs.core
     :members:
```

- Execute the script `docs/programmers/api/makehier.sh`, which automates the above. Note that the `makehier.sh` script will re-create all `.txt` files for all GC3Pie modules, so check if there were some unexpected changes (e.g., with `svn status`) before you commit!

Docstrings are written in reStructuredText format. To be able to cross-reference between differen objects in the documentation, you should be familiar with Sphinx domains in general and the Python domain in particular.

Questions?

Please write to the GC3Pie mailing list; we try to do our best to answer promptly.

2.4 List of contributors to GC3Pie

This is a list of people that have contributed to GC3Pie, in any form: be it enhancements to the code or testing out releases and new features, or simply contributing suggestions and proposing enhancements. To them all, our gratitude for trying to make GC3Pie a better tool.

The list is sorted by last name. Please send an email to <*gc3pie-dev@googlegroups.com*> for corrections.

- Tyanko Aleksiev <tyanko.alexiev@gmail.com>
- Niko Ehrenfeuchter <nikolaus.ehrenfeuchter@unibas.ch>
- Benjamin Jonen <benjamin.jonen@gmail.com>
- Sergio Maffioletti <sergio.maffioletti@gc3.uzh.ch>
- Antonio Messina <arcimboldo@gmail.com>
- Mark Monroe <markjmonroe@yahoo.com>
- Riccardo Murri <riccardo.murri@gmail.com>
- Michael Packard <mrghort@gmail.com>
- Xin Zhou <xin.zhou1983@gmail.com>

2.5 Glossary

API Acronym of *Application Programming Interface*. An API is a description of the way one piece of software asks another program to perform a service (quoted from: http://www.computerworld.com/s/article/43487/Application_Programming_Interface which see for a more detailed explanation).

Command-line The sequence of words typed at the terminal prompt in order to run a specified application.

Command-line option Arguments to a command (i.e., words on the command line) that select variants to the usual behavior of the command. For instance, a command-line option can request more verbose reporting.

Traditionally, UNIX command-line options consist of a dash (`-`), followed by one or more lowercase letters, or a double-dash (`--`) followed by a complete word or compound word.

For example, the words `-h` or `--help` usually instruct a command to print a short usage message and exit immediately after.

Core A single computing unit. This was called a *CPU* until manufacturers started packing many processing units into a single package: now the term CPU is used for the package, and *core* is one of the several independent processing units within the package.

CPU Time The total time that computing units (processor *core*) are actively executing a *job*. For single-threaded jobs, this is normally *less* then the actual duration ('wall-clock time' or *walltime*), because some time is lost in I/O and system operations. For parallel jobs the CPU time is normally larger than the duration, because

several processor cores are active on the job at the same time; the quotient of the CPU time and the duration measures the efficiency of the parallel job.

Job A computational job is a single run of a non-interactive application. The prototypical example is a run of GAMESS on a single input file.

Persistent Used in the sense of *preserved across program stops and system reboots*. In practice, it just means that the relevant data is stored on disk or in some database.

Resource Short for *computational resource*: any cluster or Grid where a job can run.

State A one-word indication of a computational *job* execution status (e.g., `RUNNING` or `TERMINATED`). The terms *state* and *status* are used interchangeably in GC3Pie documentation.

STDERR Abbreviation for "standard error stream"; it is the sequence of all text messages that a command prints to inform the user of problems or to report on operations progress. The Linux/UNIX system allows two separate output streams, one for output proper, named *STDOUT*, and STDERR for "error messages". It is entirely up to the command to tag a message as "standard output" or "standard error".

STDOUT Abbreviation for "standard output stream". It is the sequence of all characters that constitute the output of a command. The Linux/UNIX system allows two separate output streams, one for output proper, and one for "error messages", dubbed *STDERR*. It is entirely up to the command to tag a message as "standard output" or "standard error".

Session A *persistent* collection of GC3Pie tasks and jobs. Sessions are used by *The GC3Apps software* to store job status across program runs.

Walltime Short for *wall-clock time*: indicates the total running time of a *job*.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

g

A

B

C

D

E

F

G

H

I

J

K

L

S

T

U

V

W

Y