



University of  
Zurich <sup>UZH</sup>

S3IT

---

# GC3Pie basics

Riccardo Murri <[riccardo.murri@uzh.ch](mailto:riccardo.murri@uzh.ch)>

*S3IT: Services and Support for Science IT*

University of Zurich

## Concepts and glossary

## Parts of GC3Pie

GC3Pie consists of three main components:

### GC3Libs:

Python library for controlling the life-cycle of computational job collections.

### GC3Utils:

This is a small set of low-level utilities exposing the main functionality provided by GC3Libs.

### GC3Apps:

A collection of driver scripts to run large job campaigns.

## GC3Pie glossary: Application

*GC3Pie runs user applications  
on clusters and IaaS cloud resources*

An Application is just a command to execute.

## GC3Pie glossary: Application

*GC3Pie runs **user applications**  
on clusters and IaaS cloud resources*

**An Application is just a command to execute.**

If you can run it in the terminal,  
you can run it in GC3Pie.

## GC3Pie glossary: Application

*GC3Pie runs **user applications**  
on clusters and IaaS cloud resources*

An Application is just a command to execute.

A single execution of an Application  
is indeed called a Run.

(Other systems might call this a “job”.)

## GC3Pie glossary: Task

*GC3Pie **runs** user applications  
on clusters and IaaS cloud resources*

More generally, GC3Pie runs Tasks.

Tasks are a superset of applications,  
in that they include workflows.

## GC3Pie glossary: Resources

*GC3Pie runs user applications  
on clusters and IaaS cloud **resources***

**Resources are the computing infrastructures  
where GC3Pie executes applications.**

Resources include: your laptop, the “Hydra” cluster,  
the Science Cloud, Amazon AWS.



# Workflow scaffolding

## Let's start coding!

```
from gc3libs.cmdline \
    import SessionBasedScript

if __name__ == '__main__':
    import ex2a
    ex2a.AScript().run()

class AScript(SessionBasedScript):
    """
    Minimal workflow scaffolding.
    """
    def __init__(self):
        super(AScript, self).__init__(
            version='1.0')
    def new_tasks(self, extra):
        return []
```

Download this  
code into a file  
named `ex2a.py`

Open it in your  
favorite text  
editor.

## Exercise 2.A:

Download this code into a file named `ex2a.py`

1. Run the following command:

```
$ python ex2a.py --help
```

Where does the program description in the help text come from? Is there anything weird in other parts of the help text?

2. Run the following command:

```
$ python ex2a.py
```

What happens?

```
from gc3libs.cmdline \
    import SessionBasedScript
```

```
if __name__ == '__main__':
```

```
    import ex2a
```

```
    ex2a.AScript().run()
```

```
class AScript(SessionBasedScript):
```

```
    """
```

```
    Minimal workflow scaffolding.
```

```
    """
```

```
    def __init__(self):
```

```
        super(AScript, self).__init__(
            version='1.0')
```

```
    def new_tasks(self, extra):
```

```
        return []
```

These lines are  
needed in every  
session-based script.

See [issue 95](#) for  
details.

```
from gc3libs.cmdline \
    import SessionBasedScript
```

```
if __name__ == '__main__':
    import ex2a
    ex2a.AScript().run()
```

```
class AScript(SessionBasedScript):
    """
    Minimal workflow scaffolding.
    """
    def __init__(self):
        super(AScript, self).__init__(
            version='1.0')
    def new_tasks(self, extra):
        return []
```

For this to work, it is **needed** that this is the actual file name.

```

from gc3libs.cmdline \
    import SessionBasedScript

if __name__ == '__main__':
    import ex2a
    ex2a.AScript().run()

class AScript(SessionBasedScript):
    """
    Minimal workflow scaffolding.
    """
    def __init__(self):
        super(AScript, self).__init__(
            version='1.0')
    def new_tasks(self, extra):
        return []

```

This is the  
program's help text!

```

from gc3libs.cmdline \
    import SessionBasedScript

if __name__ == '__main__':
    import ex2a
    ex2a.AScript().run()

class AScript(SessionBasedScript):
    """
    Minimal workflow scaffolding.
    """
    def __init__(self):
        super(AScript, self).__init__(
            version='1.0' )
    def new_tasks(self, extra):
        return []

```

A version number  
is **mandatory**.

```

from gc3libs.cmdline \
    import SessionBasedScript

if __name__ == '__main__':
    import ex2a
    ex2a.AScript().run()

class AScript(SessionBasedScript):
    """
    Minimal workflow scaffolding.
    """
    def __init__(self):
        super(AScript, self).__init__(
            version='1.0')
    def new_tasks(self, extra):
        return []

```

**This is the core of  
the script.**

Return a list of  
Application objects,  
that GC3Pie will  
execute.



## The Application object

## Specifying commands to run, I

You need to “describe” an application to GC3Pie, in order for GC3Pie to use it.

This “description” is a blueprint from which many actual command instances can be created.

(A few such “descriptions” are already part of the core library.)

## GC3Pie application model

In GC3Pie, an application “description” is an object of the `gc3libs.Application` class (or subclasses thereof).

At a minimum: provide application-specific command-line invocation.

Advanced users can customize pre- and post-processing, react on state transitions, set computational requirements based on input files, influence scheduling. (This is standard OOP: subclass and override a method.)

## A basic example: grayscaling

```
$ convert lena.jpg -colorspace gray lena-gray.jpg
```



## Grayscale example, I

Here is how you would tell GC3Pie to run that command-line.

```
from gc3libs import Application

class GrayscaleApp(Application):
    """Convert an image file to grayscale."""
    def __init__(self, img):
        inp = basename(img)
        out = "gray-" + inp
        Application.__init__(
            self,
            arguments=[
                "convert", inp, "-colorspace", "gray", out],
            inputs=[img],
            outputs=[out],
            output_dir="grayscale.d",
            stdout="stdout.txt")
```

## Always inherit from Application

Your application class must inherit from class  
`gc3libs.Application`

```
from gc3libs import Application
```

```
class GrayscaleApp (Application) :  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The arguments parameter, I

The `arguments=` parameter is the actual command-line to be invoked.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The arguments parameter, II

The first item in the `arguments` list is the name or path to the command to run.

```
class GrayscaleApp(Application):
    """Convert an image file to grayscale."""
    def __init__(self, img):
        inp = basename(img)
        out = "gray-" + inp
        Application.__init__(
            self,
            arguments=[
                "convert", inp, "-colorspace", "gray", out],
            inputs=[img],
            outputs=[out],
            output_dir="grayscale.d",
            stdout="stdout.txt")
```



## The arguments parameter, III

The rest of the list are arguments to the program, as you would type them at the shell prompt.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out ],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The inputs parameter, I

The `inputs` parameter holds a list of files that you want to *copy* to the location where the command is executed. (Remember: this might be a remote computer!)

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The inputs parameter, II

Input files retain their name during the copy, but not the entire path.

For example:

```
inputs = [  
    '/home/rmurri/values.dat',  
    '/home/rmurri/stats.csv',  
]
```

will make files *values.dat* and *stats.csv* available in the command execution directory.

## The inputs parameter, III

You need to pass the full path name into the `inputs` list, but use only the “base name” in the command invocation.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The outputs parameter, I

The `outputs` argument list files that should be copied from the command execution directory back to your computer.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The outputs parameter, II

Output file names are *relative to the execution directory*. For example:

```
outputs = ['result.dat', 'program.log']
```

(Contrast with input files, which must be specified by *absolute path*, e.g., /home/rmurri/values.dat)

Any file with the given name that is found in the execution directory will be copied back. (*Where?* See next slides!)

If an output file is *not* found, this is *not* an error. In other words, **output files are optional**.

## The `output_dir` parameter, I

The `output_dir` parameter specifies where output files will be downloaded.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## The `output_dir` parameter, II

By default, GC3Pie does not overwrite an existing output directory: it will move the existing one to a backup name.

So, if `grayscale.d` already exists, GC3Pie will:

1. rename it to `grayscale.d.~1~`
2. create a new directory `grayscale.d`
3. download output files into the new directory



## The stdout parameter

This specifies that the command's *standard output* should be saved into a file named `stdout.txt` and retrieved along with the other output files.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt")
```

## (The stderr parameter)

There's a corresponding `stderr` option for the command's *standard error* stream.

```
class GrayscaleApp(Application):  
    """Convert an image file to grayscale."""  
    def __init__(self, img):  
        inp = basename(img)  
        out = "gray-" + inp  
        Application.__init__(  
            self,  
            arguments=[  
                "convert", inp, "-colorspace", "gray", out],  
            inputs=[img],  
            outputs=[out],  
            output_dir="grayscale.d",  
            stdout="stdout.txt",  
            stderr="stderr.txt")
```

## Mixing stdout and stderr capture

You can specify **either one** of the `stdout` and `stderr` parameters, **or both**.

If you give both, and they have the same value, then `stdout` and `stderr` will be intermixed just as they are in normal screen output.

## Let's run!

In order for a session-based script to execute something, its `new_tasks()` method must return a list of `Application` objects to run.

```
class AScript(SessionBasedScript):  
    # ...  
    def new_tasks(self, extra):  
        # 'self.param.args' is the list  
        # of command-line arguments  
        input_file = self.params.args[0]  
        app = GrayscaleApp(input_file)  
        return [app]
```

## Exercise 2.B:

Edit the `ex2a.py` file: insert the code to define the `GrayscaleApp` application, and modify the `new_tasks()` method to return one instance of it (as in the previous slide).

Can you convert the `lena.jpg` file to gray-scale using this GC3Pie script?

*(You can download the code for `GrayscaleApp` and the “Lena” image file from [this URL](#).)*

## Exercise 2.C:

Edit the script from Exercise 2.B above and add the ability to convert multiple files: for each file name given on the command line, an instance of `GrayscaleApp` should be run.

## Resource definition

# The gservers command

The `gservers` command is used to see configured and available resources.

```
> gservers
```

	localhost	
frontend	( Frontend host name )	localhost
type	( Access mode )	shellcmd
updated	( Accessible? )	True
queued	( Total queued jobs )	0
user_queued	( Own queued jobs )	0
user_run	( Own running jobs )	6
max_cores_per_job	( Max cores per job )	4
max_memory_per_core	( Max memory per core )	8GiB
max_walltime	( Max walltime per job )	8hour

Resources are defined in file `$HOME/.gc3/gc3pie.conf`

## The gservers command

The `gservers` command is used to see **configured** and available resources.

```
> gservers
```

	localhost	
frontend	( Frontend host name )	localhost
type	( Access mode )	shellcmd
updated	( Accessible? )	True
queued	( Total queued jobs )	0
user_queued	( Own queued jobs )	0
user_run	( Own running jobs )	6
max_cores_per_job	( Max cores per job )	4
max_memory_per_core	( Max memory per core )	8GiB
max_walltime	( Max walltime per job )	8hour

Resources are defined in file `$HOME/.gc3/gc3pie.conf`



## Example execution resources: local host

Allow GC3Pie to run tasks  
on the local computer.

This is the default installed  
by GC3Pie into

`$HOME/.gc3/gc3pie.conf`

```
[resource/localhost]
enabled = yes
type = shellcmd
frontend = localhost
transport = local
max_cores_per_job = 2
max_memory_per_core = 2GiB
max_walltime = 8 hours
max_cores = 2
architecture = x86_64
auth = none
override = no
```

# Example execution resources: SLURM

Allow submission of jobs to the “Hydra” cluster.

```
[resource/hydra]
enabled = no
type = slurm
frontend = login.s3it.uzh.ch
transport = ssh
auth = ssh_user_rmurri
max_walltime = 1 day
max_cores = 96
max_cores_per_job = 64
max_memory_per_core = 1 TiB
architecture = x86_64
prologue_content =
    module load cluster/largemem

[auth/ssh_user_rmurri]
type=ssh
username=rmurri
```

# Example execution resources: OpenStack

```
[resource/sciencecloud]
enabled=no
type=openstack+shellcmd
auth=openstack
```

```
vm_pool_max_size = 32
security_group_name=default
security_group_rules=
  tcp:22:22:0.0.0.0/0,
  icmp:-1:-1:0.0.0.0/0
network_ids=
  c86b320c-9542-4032-a951-c8a068894cc2
```

```
# definition of a single execution VM
instance_type=1cpu-4ram-hpc
image_id=2b227d15-8f6a-42b0-b744-ede52ebe59f7
```

```
max_cores_per_job = 8
max_memory_per_core = 4 GiB
max_walltime = 90 days
max_cores = 32
architecture = x86_64
```

```
# how to connect
vm_auth=ssh_user_ubuntu
keypair_name=rmurri
public_key=~/.ssh/id_dsa.pub
```

```
[auth/ssh_user_ubuntu]
# default user on Ubuntu VM images
type=ssh
username=ubuntu
```

```
[auth/openstack]
# only need to set the 'type' here;
# any other value will be taken from
# the 'OS_*' environment variables
type = openstack
```

Allow running tasks on the  
“ScienceCloud” VM  
infrastructure.

**Exercise 2.D:** Change the configuration file

`~/ .gc3/gc3pie.conf` to enable the `sciencecloud` resource. Verify with the `gservers` command that it works.

**Exercise 2.E:** Run the grayscale converter on Science Cloud. Do you need to change anything in the code?