# DRMAA Python Documentation
## *Release 0.7.8*

**Dan Blanchard, David Ressman, Enrico Sirola**

**Jul 03, 2017**

# Contents

Contents:

## Distributed Resource Management Application API

This guide is a tutorial for getting started programming with DRMAA. It is basically a one to one translation of the original in C for Grid Engine. It assumes that you already know what DRMAA is and that you have drmaa-python installed. If not, have a look at Installing. The following code segments are also included in the repository.

## Starting and Stopping a Session

The following two code segments show the most basic DRMAA python binding program.

Listing 1.1: example1.py

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

import drmaa

def main():
    """Create a drmaa session and exit"""
    with drmaa.Session() as s:
        print('A session was started successfully')

if __name__=='__main__':
    main()
```

The first thing to notice is that every call to a DRMAA function will return an error code. In this tutorial, we ignore all error codes.

Now let's look at the functions being called. First, on line 7, we initialise a `Session` object by calling `drmaa.Session()`. The `Session` is automatically initialized via `initialize()`, and it creates a session and starts an event client listener thread. The session is used for organizing jobs submitted through DRMAA, and the thread is used to receive updates from the queue master about the state of jobs and the system in general. Once `initialize()` has been called successfully, it is the responsibility of the calling application to also call `exit()` before terminating, but since we're using it as a context manager via the `with` statement, that will happene automatically. If an application

does not call `exit()` before terminating, session state may be left behind in the user's home directory, and the queue master may be left with a dead event client handle, which can decrease queue master performance.

At the end of our program, `exit()` is called automatically when we leave the `with` block. `exit()` cleans up the session and stops the event client listener thread. Most other DRMAA functions must be called before `exit()`. Some functions, like `getContact()`, can be called after `exit()`, but these functions only provide general information. Any function that does work, such as `runJob()` or `wait()` must be called before `exit()` is called. If such a function is called after `exit()` is called, it will return an error.

Listing 1.2: example1.1.py

```python
1   #!/usr/bin/env python
2
3   import drmaa
4
5   def main():
6       """
7       Create a session, show that each session has an ID, use session ID to
8       disconnect, then reconnect. Finally, exit.
9       """
10      s = drmaa.Session()
11      s.initialize()
12      print('A session was started successfully')
13      response = s.contact
14      print('session contact returns: %s' % response)
15      s.exit()
16      print('Exited from session')
17
18      s.initialize(response)
19      print('Session was restarted successfullly')
20      s.exit()
21
22
23   if __name__=='__main__':
24       main()
```

This example is very similar to *Example 1*. The difference is that it uses the Grid Engine feature of reconnectable sessions. The DRMAA concept of a session is translated into a session tag in the Grid Engine job structure. That means that every job knows to which session it belongs. With reconnectable sessions, it's possible to initialize the DRMAA library to a previous session, allowing the library access to that session's job list. The only limitation, though, is that jobs which end between the calls to `exit()` and `init()` will be lost, as the reconnecting session will no longer see these jobs, and so won't know about them.

On line 13, we use the contact attribute to get the contact information for this session. On line 15 we then exit the session. On line 18, we use the stored contact information to reconnect to the previous session. Had we submitted jobs before calling `exit()`, those jobs would now be available again for operations such as `wait()` and `synchronize()`. Finally, on line 19 we exit the session a second time.

## Running a Job

The following code segments show how to use the DRMAA python binding to submit a job to Grid Engine. It submits a small shell script (`sleeper.sh`) which takes two arguments:

```bash
#!/bin/bash
echo "Hello world, the answer is $1"
```

```
sleep 3s
echo "$2 Bye world!"
```

Listing 1.3: example2.py

```python
1  #!/usr/bin/env python
2
3  import drmaa
4  import os
5
6  def main():
7      """
8      Submit a job.
9      Note, need file called sleeper.sh in current directory.
10     """
11     with drmaa.Session() as s:
12         print('Creating job template')
13         jt = s.createJobTemplate()
14         jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
15         jt.args = ['42', 'Simon says:']
16         jt.joinFiles=True
17
18         jobid = s.runJob(jt)
19         print('Your job has been submitted with ID %s' % jobid)
20
21         print('Cleaning up')
22         s.deleteJobTemplate(jt)
23
24  if __name__=='__main__':
25      main()
```

The beginning and end of this program are the same as the first example. What's new is in lines 12–22. On line 12 we ask DRMAA to allocate a job template for us. A job template is a structure used to store information about a job to be submitted. The same template can be reused for multiple calls to `runJob()` or `runBulkJob()`.

On line 13 we set the `remoteCommand` attribute. This attribute tells DRMAA where to find the program we want to run. Its value is the path to the executable. The path can be either relative or absolute. If relative, it is relative to the `workingDirectory` attribute, which if not set defaults to the user's home directory. For more information on DRMAA attributes, please see the attributes man page. Note that for this program to work, the script `sleeper.sh` must be in the current directory.

On line 14 we set the `args` attribute. This attribute tells DRMAA what arguments to pass to the executable.

On line 17 we submit the job with `runJob()`. DRMAA will place the ID assigned to the job into the character array we passed to `runJob()`. The job is now running as though submitted by `qsub` or `bsub`. At this point calling `exit()` and/or terminating the program will have no effect on the job.

To clean things up, we delete the job template on line 21. This frees the memory DRMAA set aside for the job template, but has no effect on submitted jobs.

If instead of a single job we had wanted to submit an array job, we could have replaced the else on line 18 and 19 with the following:

Listing 1.4: example2.1.py

```
jobid = s.runBulkJobs(jt, 1, 30, 2)
print('Your jobs have been submitted with IDs %s' % jobid)
```

This code segment submits an array job with 15 tasks numbered 1, 3, 5, 7, etc. An important difference to note is that `runBulkJobs()` returns the job ids in a list. On the last line, we print all the job ids.

## Waiting for a Job

Now we're going to extend our example to include waiting for a job to finish.

Listing 1.5: example3.py

```python
1   #!/usr/bin/env python
2
3   import drmaa
4   import os
5
6   def main():
7       """
8       Submit a job and wait for it to finish.
9       Note, need file called sleeper.sh in home directory.
10      """
11      with drmaa.Session() as s:
12          print('Creating job template')
13          jt = s.createJobTemplate()
14          jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
15          jt.args = ['42', 'Simon says:']
16          jt.joinFiles = True
17
18          jobid = s.runJob(jt)
19          print('Your job has been submitted with ID %s' % jobid)
20
21          retval = s.wait(jobid, drmaa.Session.TIMEOUT_WAIT_FOREVER)
22          print('Job: {0} finished with status {1}'.format(retval.jobId, retval.
    →hasExited))
23
24          print('Cleaning up')
25          s.deleteJobTemplate(jt)
26
27  if __name__=='__main__':
28      main()
```

This example is very similar to *Example 2* except for line 21. On line 21 we call `wait()` to wait for the job to end. We have to give `wait()` both the ID of the job for which we want to wait, and also how long we are willing to wait for the job to finish. This could be a number of seconds, or it could be either `TIMEOUT_WAIT_FOREVER` or `TIMEOUT_NO_WAIT`. `wait()` returns a `JobInfo` tuple, which has the following attributes: `jobId`, `hasExited`, `hasSignal`, `terminatedSignal`, `hasCoreDump`, `wasAborted`, `exitStatus`, and `resourceUsage`. `jobId` is particularly useful if we passed in `JOB_IDS_SESSION_ANY` as the ID argument for `wait()`, because without it we would have no way of knowing which job it actually waited for. Lastly, we print out the job ID and the exit status on line 22.

An alternative to `wait()` when working with multiple jobs, such as jobs submitted by `runBulkJobs()` or multiple calls to `runJob()` is `synchronize()`. `synchronize()` waits for a set of jobs to finish. To use

`synchronize()`, we could replace lines 18–22 with the following:

Listing 1.6: example3.1.py

```python
joblist = s.runBulkJobs(jt, 1, 30, 2)
print('Your jobs have been submitted with IDs %s' % joblist)

s.synchronize(joblist, drmaa.Session.TIMEOUT_WAIT_FOREVER, True)
```

With the switch to `runBulkJobs()` we now have several jobs for which to wait. On line 20, instead of calling `wait()`, we call `synchronize()`. `synchronize()` takes only three interesting parameters. The first is the list of string IDs for which to wait. If the special ID, `JOB_IDS_SESSION_ALL`, appears in the array, `synchronize()` will wait for all jobs submitted via DRMAA during this session (i.e., since `initialize()` was called). The second is how long to wait for all the jobs in the list to finish. This is the same as the timeout parameter for `wait()`. The third is whether this call to `synchronize()` should clean up after the job. After a job completes, it leaves behind accounting information, such as exit status and usage, until either `wait()` or `synchronize()` with `dispose` set to `True` is called. It is the responsibility of the application to make sure one of these two functions is called for every job. Not doing so creates a memory leak. Note that calling `synchronize()` with dispose set to true flushes all accounting information for all jobs in the list. If you want to use `synchronize()` and still recover the accounting information, set `dispose` to `False` and call `wait()` for each job. To do this in *Example 3*, we would replace lines 18–22 with the following:

Listing 1.7: example3.2.py

```python
joblist = s.runBulkJobs(jt, 1, 30, 2)
print('Your jobs have been submitted with IDs %s' % joblist)

s.synchronize(joblist, drmaa.Session.TIMEOUT_WAIT_FOREVER, False)
for curjob in joblist:
    print('Collecting job ' + curjob)
    retval = s.wait(curjob, drmaa.Session.TIMEOUT_WAIT_FOREVER)
    print('Job: {0} finished with status {1}'.format(retval.jobId,
                                                      retval.hasExited))
```

What's different is that on line 21 we set `dispose` to `False`, and then on lines 22–26 we wait once for each job, printing the exit status and usage information as we did in *Example 3*.

We pass `joblist` to `synchronize()` to wait for each job specifically. Otherwise, the `wait()` could end up waiting for a job submitted after the call to `synchronize()`.

## Controlling a Job

Now let's look at an example of how to control a job from DRMAA:

Listing 1.8: example4.py

```python
1  #!/usr/bin/env python
2
3  import drmaa
4  import os
5
6  def main():
7      """Submit a job, then kill it.
8      Note, need file called sleeper.sh in home directory.
9      """
10     with drmaa.Session() as s:
```

```
11          print('Creating job template')
12          jt = s.createJobTemplate()
13          jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
14          jt.args = ['42', 'Simon says:']
15          jt.joinFiles = True
16
17          jobid = s.runJob(jt)
18          print('Your job has been submitted with ID %s' % jobid)
19          # options are: SUSPEND, RESUME, HOLD, RELEASE, TERMINATE
20          s.control(jobid, drmaa.JobControlAction.TERMINATE)
21
22          print('Cleaning up')
23          s.deleteJobTemplate(jt)
24
25  if __name__=='__main__':
26      main()
```

This example is very similar to *Example 2* except for line 20. On line 20 we use `control()` to delete the job we just submitted. Aside from deleting the job, we could have also used `control()` to suspend, resume, hold, or release it.

Note that `control()` can be used to control jobs not submitted through DRMAA. Any valid SGE job ID could be passed to `control()` as the ID of the job to delete.

## Getting Job Status

Here's an example of using DRMAA to query the status of a job:

Listing 1.9: example5.py

```python
1   #!/usr/bin/env python
2
3   import drmaa
4   import time
5   import os
6
7   def main():
8       """
9       Submit a job, and check its progress.
10      Note, need file called sleeper.sh in home directory.
11      """
12      with drmaa.Session() as s:
13          print('Creating job template')
14          jt = s.createJobTemplate()
15          jt.remoteCommand = os.path.join(os.getcwd(), 'sleeper.sh')
16          jt.args = ['42', 'Simon says:']
17          jt.joinFiles=True
18
19          jobid = s.runJob(jt)
20          print('Your job has been submitted with ID %s' % jobid)
21
22          # Who needs a case statement when you have dictionaries?
23          decodestatus = {drmaa.JobState.UNDETERMINED: 'process status cannot be
    ↪determined',
24                          drmaa.JobState.QUEUED_ACTIVE: 'job is queued and active',
25                          drmaa.JobState.SYSTEM_ON_HOLD: 'job is queued and in system
    ↪hold',
```

```
26                          drmaa.JobState.USER_ON_HOLD: 'job is queued and in user hold',
27                          drmaa.JobState.USER_SYSTEM_ON_HOLD: 'job is queued and in
    ↪user and system hold',
28                          drmaa.JobState.RUNNING: 'job is running',
29                          drmaa.JobState.SYSTEM_SUSPENDED: 'job is system suspended',
30                          drmaa.JobState.USER_SUSPENDED: 'job is user suspended',
31                          drmaa.JobState.DONE: 'job finished normally',
32                          drmaa.JobState.FAILED: 'job finished, but failed'}
33
34          for ix in range(10):
35              print('Checking %s of 10 times' % ix)
36              print decodestatus(s.jobStatus(jobid))
37              time.sleep(5)
38
39          print('Cleaning up')
40          s.deleteJobTemplate(jt)
41
42 if __name__=='__main__':
43     main()
```

Again, this example is very similar to *Example 2*, this time with the exception of lines 22–40. On line 36, we use
jobStatus() to get the status of the job. Line 43 determine what the job status is and report it.

# Getting DRM information

Lastly, let's look at how to query the DRMAA library for information about the DRMS and the DRMAA implementation itself:

Listing 1.10: example6.py

```
1  #!/usr/bin/env python
2
3  import drmaa
4
5  def main():
6      """ Query the system. """
7      with drmaa.Session() as s:
8          print('A DRMAA object was created')
9          print('Supported contact strings: %s' % s.contact)
10         print('Supported DRM systems: %s' % s.drmsInfo)
11         print('Supported DRMAA implementations: %s' % s.drmaaImplementation)
12         print('Version %s' % s.version)
13
14         print('Exiting')
15
16 if __name__=='__main__':
17     main()
```

On line 9, we get the contact string list. This is the list of contact strings that will be understood by this DRMAA
instance. Normally on of these strings is used to select to which DRM this DRMAA instance should be bound. On line
10, we get the list of supported DRM systems. On line 11, we get the list of supported DRMAA implementations. On
line 12, we get the version number of the DRMAA C binding specification supported by this DRMAA implementation.

# drmaa Package

A python package for DRM job submission and control.

This package is an implementation of the DRMAA 1.0 Python language binding specification ([http://www.ogf.org/documents/GFD.143.pdf](http://www.ogf.org/documents/GFD.143.pdf)). The source is hosted on GitHub: [https://github.com/pygridtools/drmaa-python](https://github.com/pygridtools/drmaa-python) Releases are available from PyPi: [https://pypi.python.org/pypi/drmaa](https://pypi.python.org/pypi/drmaa) Documentation is hosted on ReadTheDocs: [http://drmaa-python.readthedocs.org/](http://drmaa-python.readthedocs.org/)

> **author** Enrico Sirola ([enrico.sirola@statpro.com](mailto:enrico.sirola@statpro.com))
>
> **author** Dan Blanchard ([dan.blanchard@gmail.com](mailto:dan.blanchard@gmail.com))

**class** drmaa.**JobInfo** (*jobId*, *hasExited*, *hasSignal*, *terminatedSignal*, *hasCoreDump*, *wasAborted*, *exitStatus*, *resourceUsage*)

Bases: `tuple`

**exitStatus**
> Alias for field number 6

**hasCoreDump**
> Alias for field number 4

**hasExited**
> Alias for field number 1

**hasSignal**
> Alias for field number 2

**jobId**
> Alias for field number 0

**resourceUsage**
> Alias for field number 7

**terminatedSignal**
> Alias for field number 3

**wasAborted**
> Alias for field number 5

**class** drmaa.**JobTemplate**(*\*\*kwargs*)
Bases: object

A job to be submitted to the DRM.

**HOME_DIRECTORY = u'$drmaa_hd_ph$'**

**PARAMETRIC_INDEX = u'$drmaa_incr_ph$'**

**WORKING_DIRECTORY = u'$drmaa_wd_ph$'**

**args**
A DRMAA attribute representing a list.

To be managed with vector C DRMAA attribute management functions.

**attributeNames**
The list of supported DRMAA scalar attribute names.

This is apparently useless now, and should probably substituted by the list of attribute names of the JobTemplate instances.

**blockEmail = False**

**deadlineTime = u''**

**delete**()
Deallocate the underlying DRMAA job template.

**email**
A DRMAA attribute representing a list.

To be managed with vector C DRMAA attribute management functions.

**errorPath = u''**

**hardRunDurationLimit**
A DRMAA attribute, to be managed with scalar C DRMAA attribute management functions.

**hardWallclockTimeLimit**
A DRMAA attribute, to be managed with scalar C DRMAA attribute management functions.

**inputPath = u''**

**jobCategory = u''**

**jobEnvironment**
A DRMAA attribute representing a python dict.

To be managed with vector C DRMAA attribute management functions.

**jobName = u''**

**jobSubmissionState = u''**

**joinFiles = False**

**nativeSpecification = u''**

**outputPath = u''**

**remoteCommand = u''**

**softRunDurationLimit**
A DRMAA attribute, to be managed with scalar C DRMAA attribute management functions.

**softWallclockTimeLimit**
A DRMAA attribute, to be managed with scalar C DRMAA attribute management functions.

**startTime = u''**

**transferFiles = u''**

**workingDirectory = u''**

class drmaa.**Session**(*contactString=None*)

Bases: object

The DRMAA Session.

This class is the entry point for communicating with the DRM system

**JOB_IDS_SESSION_ALL = 'DRMAA_JOB_IDS_SESSION_ALL'**

**JOB_IDS_SESSION_ANY = 'DRMAA_JOB_IDS_SESSION_ANY'**

**TIMEOUT_NO_WAIT = 0**

**TIMEOUT_WAIT_FOREVER = -1**

**contact = u''**

static **control**(*jobId*, *operation*)

Used to hold, release, suspend, resume, or kill the job identified by jobId.

> **Parameters**
>
>> **jobId** [string] if jobId is *Session.JOB_IDS_SESSION_ALL* then this routine acts on all jobs submitted during this DRMAA session up to the moment control() is called. The legal values for action and their meanings are
>>
>> **operation** [string]
>>
>> **possible values are:**
>>
>>> *JobControlAction.SUSPEND* stop the job
>>>
>>> *JobControlAction.RESUME* (re)start the job
>>>
>>> *JobControlAction.HOLD* put the job on-hold
>>>
>>> *JobControlAction.RELEASE* release the hold on the job
>>>
>>> *JobControlAction.TERMINATE* kill the job

To avoid thread races in multithreaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission calls or control calls that may change the number of remote jobs.

This method returns once the action has been acknowledged by the DRM system, but does not necessarily wait until the action has been completed. Some DRMAA implementations may allow this method to be used to control jobs submitted external to the DRMAA session, such as jobs submitted by other DRMAA session in other DRMAA implementations or jobs submitted via native utilities.

static **createJobTemplate**()

Allocates a new job template.

The job template is used to set the environment for jobs to be submitted. Once the job template has been created, it should also be deleted (via deleteJobTemplate()) when no longer needed. Failure to do so may result in a memory leak.

static **deleteJobTemplate**(*jobTemplate*)

Deallocate a job template.

> **Parameters**

> **jobTemplate** [JobTemplate] the job temptare to be deleted

This routine has no effect on running jobs.

**drmaaImplementation = u''**

**drmsInfo = u''**

static **exit**()
> Used to disengage from DRM.

> This routine ends the current DRMAA session but doesn't affect any jobs (e.g., queued and running jobs remain queued and running). exit() should be called only once, by only one of the threads. Additional calls to exit() beyond the first will throw a NoActiveSessionException.

static **initialize**(*contactString=None*)
> Used to initialize a DRMAA session for use.

> > **Parameters**

> > > **contactString** [string or None] implementation-dependent string that may be used to specify which DRM system to use

> This method must be called before any other DRMAA calls. If contactString is None, the default DRM system is used, provided there is only one DRMAA implementation available. If there is more than one DRMAA implementation available, initialize() throws a NoDefaultContactStringSelectedException. initialize() should be called only once, by only one of the threads. The main thread is recommended. A call to initialize() by another thread or additional calls to initialize() by the same thread with throw a SessionAlreadyActiveException.

static **jobStatus**(*jobId*)
> returns the program status of the job identified by jobId.

> The possible values returned from this method are:

> > •*JobState.UNDETERMINED*: process status cannot be determined,

> > •*JobState.QUEUED_ACTIVE*: job is queued and active,

> > •*JobState.SYSTEM_ON_HOLD*: job is queued and in system hold,

> > •*JobState.USER_ON_HOLD*: job is queued and in user hold,

> > •*JobState.USER_SYSTEM_ON_HOLD*: **job is queued and in user and** system hold,

> > •*JobState.RUNNING*: job is running,

> > •*JobState.SYSTEM_SUSPENDED*: job is system suspended,

> > •*JobState.USER_SUSPENDED*: job is user suspended,

> > •*JobState.DONE*: job finished normally, and

> > •*JobState.FAILED*: job finished, but failed.

> The DRMAA implementation should always get the status of the job from the DRM system unless the status has already been determined to be FAILED or DONE and the status has been successfully cached. Terminated jobs return a FAILED status.

static **runBulkJobs**(*jobTemplate*, *beginIndex*, *endIndex*, *step*)
> Submit a set of parametric jobs, each with attributes defined in the job template.

> > **Parameters**

> > > **jobTemplate** [JobTemplate] the template representng jobs to be run

> > > **beginIndex** [int] index of the first job

---

> > **endIndex** [int] index of the last job
>
> > **step** [int] the step between job ids

The returned job identifiers are Strings identical to those returned from the underlying DRM system. The JobTemplate class defines a *JobTemplate.PARAMETRIC_INDEX* placeholder for use in specifying paths. This placeholder is used to represent the individual identifiers of the tasks submitted through this method.

static **runJob** (*jobTemplate*)
> Submit a job with attributes defined in the job template.

> > **Parameters**

> > > **jobTemplate** [JobTemplate] the template representing the job to be run

> The returned job identifier is a String identical to that returned from the underlying DRM system.

static **synchronize** (*jobIds*, *timeout=-1*, *dispose=False*)
> Waits until all jobs specified by jobList have finished execution.

> > **Parameters**

> > > **jobIds** If jobIds contains *Session.JOB_IDS_SESSION_ALL*, then this method waits for all jobs submitted during this DRMAA session up to the moment synchronize() is called

> > > **timeout** [int] maximum time (in seconds) to be waited for the completion of a job.

> > > > The value *Session.TIMEOUT_WAIT_FOREVER* may be specified to wait indefinitely for a result. The value *Session.TIMEOUT_NO_WAIT* may be specified to return immediately if no result is available.

> > > **dispose** [bool] specifies how to treat the reaping of the remote job's internal data record, which includes a record of the job's consumption of system resources during its execution and other statistical information. If set to True, the DRM will dispose of the job's data record at the end of the synchronize() call. If set to False, the data record will be left for future access via the wait() method. It is the responsibility of the application to make sure that either *synchronize()* or 'wait()'is called for every job. Not doing so creates a memory leak. Note that calling synchronize() with dispose set to true flushes all accounting information for all jobs in the list.

> To avoid thread race conditions in multithreaded applications, the DRMAA implementation user should explicitly synchronize this call with any other job submission calls or control calls that may change the number of remote jobs.

> If the call exits before the timeout has elapsed, all the jobs have been waited on or there was an interrupt. If the invocation exits on timeout, an ExitTimeoutException is thrown. The caller should check system time before and after this call in order to be sure of how much time has passed.

**version** = Version(major=10L, minor=10L)

static **wait** (*jobId*, *timeout=-1*)
> Wait for a job with jobId to finish execution or fail.

> > **Parameters**

> > > *jobId* [str] The job id to wait completion for.

> > > > If the special string, *Session.JOB_IDS_SESSION_ANY*, is provided as the jobId, this routine will wait for any job from the session

> > > *timeout* [float] The timeout value is used to specify the desired behavior when a result is not immediately available.

> The value *Session.TIMEOUT_WAIT_FOREVER* may be specified to wait indefinitely
> for a result. The value *Session.TIMEOUT_NO_WAIT* may be specified to return im-
> mediately if no result is available. Alternatively, a number of seconds may be specified
> to indicate how long to wait for a result to become available

This routine is modeled on the wait3 POSIX routine. If the call exits before timeout, either the job has
been waited on successfully or there was an interrupt. If the invocation exits on timeout, an *ExitTimeou-*
*tException* is thrown. The caller should check system time before and after this call in order to be sure
how much time has passed. The routine reaps job data records on a successful call, so any subsequent
calls to wait() will fail, throwing an *InvalidJobException*, meaning that the job's data record has been
already reaped. This exception is the same as if the job were unknown. (The only case where wait() can
be successfully called on a single job more than once is when the previous call to wait() timed out before
the job finished.)

**exception** drmaa.**AlreadyActiveSessionException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**AuthorizationException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**ConflictingAttributeValuesException**
    Bases: drmaa.errors.DrmaaException, exceptions.AttributeError

**exception** drmaa.**DefaultContactStringException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**DeniedByDrmException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**DrmCommunicationException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**DrmsExitException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**DrmsInitException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**ExitTimeoutException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**HoldInconsistentStateException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**IllegalStateException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**InternalException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**InvalidAttributeFormatException**
    Bases: drmaa.errors.DrmaaException, exceptions.AttributeError

**exception** drmaa.**InvalidContactStringException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**InvalidJobException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**InvalidJobTemplateException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**NoActiveSessionException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**NoDefaultContactStringSelectedException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**ReleaseInconsistentStateException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**ResumeInconsistentStateException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**SuspendInconsistentStateException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**TryLaterException**
    Bases: drmaa.errors.DrmaaException

**exception** drmaa.**UnsupportedAttributeException**
    Bases: drmaa.errors.DrmaaException, exceptions.AttributeError

**exception** drmaa.**InvalidArgumentException**
    Bases: drmaa.errors.DrmaaException, exceptions.AttributeError

**exception** drmaa.**InvalidAttributeValueException**
    Bases: drmaa.errors.DrmaaException, exceptions.AttributeError

**exception** drmaa.**OutOfMemoryException**
    Bases: drmaa.errors.DrmaaException, exceptions.MemoryError

drmaa.**control_action_to_string**(*code*)

drmaa.**job_state**(*code*)

**class** drmaa.**JobControlAction**
    Bases: object

    **HOLD = u'hold'**

    **RELEASE = u'release'**

    **RESUME = u'resume'**

    **SUSPEND = u'suspend'**

    **TERMINATE = u'terminate'**

**class** drmaa.**JobState**
    Bases: object

    **DONE = u'done'**

    **FAILED = u'failed'**

    **QUEUED_ACTIVE = u'queued_active'**

    **RUNNING = u'running'**

    **SYSTEM_ON_HOLD = u'system_on_hold'**

    **SYSTEM_SUSPENDED = u'system_suspended'**

    **UNDETERMINED = u'undetermined'**

    **USER_ON_HOLD = u'user_on_hold'**

    **USER_SUSPENDED = u'user_suspended'**

> **USER_SYSTEM_ON_HOLD = u'user_system_on_hold'**
>
> **USER_SYSTEM_SUSPENDED = u'user_system_suspended'**

**class** drmaa.**JobSubmissionState**

> Bases: object
>
> **ACTIVE_STATE = u'drmaa_active'**
>
> **HOLD_STATE = u'drmaa_hold'**

drmaa.**status_to_string**(*status*)

drmaa.**string_to_control_action**(*operation*)

drmaa.**submission_state**(*code*)

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## d

# Index