Next Up Previous Contents Index
Next: 7. Platform-Specific Information **Up:** 6. Application Programming Interfaces **Previous:** 6.6 The HTCondor Perl **Contents** **Index**

Subsections

6.7 Python Bindings

The Python module provides bindings to the client-side APIs for HTCondor and the ClassAd language.

These Python bindings depend on loading the HTCondor shared libraries; this means the same code is used here as the HTCondor client tools. It is more efficient in terms of memory and CPU to utilize these bindings than to parse the output of the HTCondor client tools when writing applications in Python.

6.7.1 `htcondor` Module

The `htcondor` module provides a client interface to the various HTCondor daemons. It tries to provide functionality similar to the HTCondor command line tools.

`htcondor` module functions:

`platform( )` Returns the platform of HTCondor this module is running on.
`version( )` Returns the version of HTCondor this module is linked against.
`reload_config( )` Reload the HTCondor configuration from disk.
`send_command( ad, (DaemonCommands)dc, (str)target = None)` Send a command to an HTCondor daemon specified by a location ClassAd. `ad` is a ClassAd specifying the location of the daemon; typically, found by using `Collector.locate(...)`. `dc` is a command type; must be a member of the enum `DaemonCommands`. `target` is an optional parameter, representing an additional command to send to a daemon. Some commands require additional arguments; for example, sending `DaemonOff` to a *condor_master* requires one to specify which subsystem to turn off.
`read_events( file_obj, is_xml = True )` Read and parse an HTCondor event log file. Returns a Python iterator of ClassAds. Parameter `file_obj` is a file object corresponding to an HTCondor event log. The optional parameter `is_xml` specifies whether the event log is XML-formatted.
`send_alive( ad, pid, timeout )` Send a keep alive message to an HTCondor daemon. Parameter `ad` is a ClassAd specifying the location of the daemon. This ClassAd is typically found by using `Collector.locate(...)`.

Parameter `pid` is the process identifier for the keep alive. The default value of `None` uses the value from `os.getpid()`.

Parameter `timeout` is the number of seconds that this keep alive is valid. If a new keep alive is not received by the *condor_master* in time, then the process will be terminated. The default value is controlled by configuration variable `NOT_RESPONDING_TIMEOUT`.

`set_subsystem( name, type = Auto )`

Set the subsystem name for the object.

Parameter `name` is the subsystem name.

Parameter `type` is the HTCondor daemon type, taken from the `SubsystemType` enum. The default value of `Auto` infers the type from the `name` parameter.

`lock( file_obj, lock_type )`

Take a lock on a file object using the HTCondor locking protocol, which is distinct from typical POSIX locks. Returns a context manager object; the lock is released as this context manager object is destroyed.

Parameter `file_obj` is a file object corresponding to the file which should be locked.

Parameter `lock_type` specifies the string `"ReadLock"` if the lock should be for reads or `"WriteLock"` if the lock should be for writes.

`enable_debug( )`

Enable debugging output from HTCondor, where output is sent to `stderr`. The logging level is controlled by `TOOL_DEBUG`.

`enable_log( )`

Enable debugging output from HTCondor, where output is sent to a file. The log level is controlled by `TOOL_DEBUG`, and the file used is controlled by `TOOL_LOG`.

`log( level, msg )` Log a message to the HTCondor logging subsystem.

Parameter `level` is the Log category and formatting indicator. Use the `LogLevel` enum to get list of attributes that may be OR'd together.

Parameter `msg` is a String message to log.

`poll( active_queries )`

Wait on the results of multiple query iteratories. Param `active_queries` is a list of query iterators as returned by `xquery()`.

This function returns an iterator which yields the next ready query iterator. The returned iterator stops when all results have been consumed for all iterators.

The iterator returned by `xquery` has a method named `nextAdsNonBlocking` which returns a list of all ads available without blocking.

The module object, `param`, is a dictionary-like object providing access to the configuration variables in the current HTCondor configuration.

The `Schedd` class:

`__init__( classad )`

Create an instance of the `Schedd` class.

Optional parameter `classad` describes the location of the remote *condor_schedd* daemon. If the parameter is omitted, the local *condor_schedd* daemon is used.

`transaction( flags = 0, continue_txn = False )`

Start a transaction with the *condor_schedd*. Returns a transaction context manager. Starting a new transaction while one is ongoing is an error.

The optional parameter `flags` defaults to 0. Transaction flags are from the the enum `htcondor.TransactionFlags`, and the three flags are `NonDurable`, `SetDirty`, or `ShouldLog`. `NonDurable` is used for performance, as it eliminates extra `fsync()` calls. If the *condor_schedd* crashes before the transaction is written to disk, the transaction will be retried on restart of the *condor_schedd*. `SetDirty` marks the changed ClassAds as dirty, so an update notification is sent to the *condor_shadow* and the *condor_gridmanager*. `ShouldLog` causes changes to the job queue to be logged in the job event log file.

The optional parameter `continue_txn` defaults to `false`; set the value to `true` to extend an ongoing transaction.

`act( (JobAction)action, (object)job_spec )`

Change status of job(s) in the *condor_schedd* daemon. The integer return value is a `ClassAd` object describing the number of jobs changed.

Parameter `action` is the action to perform; must be of the enum `JobAction`.

Parameter `job_spec` is the job specification. It can either be a list of job IDs or a string specifying a constraint to match jobs.

`edit( (object)job_spec, (str)attr, (object)value )`

Edit one or more jobs in the queue.

Parameter `job_spec` is either a list of jobs, with each given as `ClusterId.ProcId` or a string containing a constraint to match jobs against.

Parameter `attr` is the attribute name of the attribute to edit.

Parameter `value` is the new value of the job attribute. It should be a string, which will be converted to a ClassAd expression, or an `ExprTree` object.

`query( constraint = true, attr_list = [] )`

Query the *condor_schedd* daemon for jobs. Returns a list of ClassAds representing the matching jobs, containing at least the requested attributes requested by the second parameter.

The optional parameter `constraint` provides a constraint for filtering out jobs. It defaults to `True`.

Parameter `attr_list` is a list of attributes for the *condor_schedd* daemon to project along. It defaults to having the *condor_schedd* daemon return all attributes.

`xquery( constraint = true, attr_list = [], limit, opts, name )`

Query the *condor_schedd* daemon for jobs. Returns an iterator of ClassAds representing the matching jobs containing at least the list of attributes requested by the second parameter.

The optional parameter `constraint` provides a constraint for filtering out jobs. It defaults to `True`.

Parameter `attr_list` is a list of attributes for the *condor_schedd* daemon to project along. It defaults to having the *condor_schedd* daemon return all attributes.

Parameter `limit` is the maximum number of results this query will

return.

Parameter `opts` specifies any additional query options. Currently, the only non-default option is `QueryOpts.AutoCluster`, which returns autoclusters in the schedd, not jobs.

Parameter `name` provides a *tag* name for the returned query iterator. This string will always be returned from the `tag()` method of the returned iterator. The default value is the *condor_schedd*'s name. This tag is useful to identify different queries when using the `poll()` module function.

`history( (object) requirements, (list) projection, (int) match )`

Request history records from the *condor_schedd* daemon. Returns an iterator to a set of ClassAds representing completed jobs.

Parameter `requirements` is either an `ExprTree` or a string that can be parsed as an expression. The expression represents the requirements that all returned jobs should match.

Parameter `projection` is a list of all the ClassAd attributes that are to be included for each job. The empty list causes all attributes to be included.

Parameter `match` is an integer cap on the number of jobs to include.

`submit( ad, count = 1, spool = false, ad_results = None )`

Submit one or more jobs to the *condor_schedd* daemon. Returns the newly created cluster ID.

This method requires the invoker to provide a ClassAd for the new job cluster; such a ClassAd contains attributes with different names than the commands in a submit description file. As an example, the stdout file is referred to as `output` in the submit description file, but `Out` in the ClassAd. To generate an example ClassAd, take a sample submit description file and invoke

`condor_submit -dump <filename> [cmdfile]`

Then, load the resulting contents of <filename> into Python.

Parameter `ad` is the ClassAd describing the job cluster.

Parameter `count` is the number of jobs to submit to the cluster. Defaults to 1.

Parameter `spool` inserts the necessary attributes into the job for it to have the input files spooled to a remote *condor_schedd* daemon. This parameter is necessary for jobs submitted to a remote *condor_schedd*.

Parameter `ad_results`, if set to a list, will contain the job ClassAds resulting from the job submission. These are useful for interacting with the job spool at a later time.

`submitMany( cluster_ad, proc_ads, spool = false, ad_results = None )`

Submit multiple jobs to the *condor_schedd* daemon, possibly including several distinct processes. Returns the newly created cluster ID.

This method requires the invoker to provide a ClassAd, `cluster_ad` for the new job cluster; this is the same format as in the `submit()` method.

The `proc_ads` parameter is a list of 2-tuples; each tuple has the format of `(proc_ad, count)`. For each list entry, this will result in `count` jobs being submitted inheriting from both `cluster_ad` and `proc_ad`.

Parameter `spool` inserts the necessary attributes into the job for it to have the input files spooled to a remote *condor_schedd* daemon.

This parameter is necessary for jobs submitted to a remote *condor_schedd*.

Parameter `ad_results`, if set to a list, will contain the job ClassAds resulting from the job submission. These are useful for interacting with the job spool at a later time.

`spool( ad_list )`

Spools the files specified in a list of job ClassAds to the *condor_schedd*. Throws a RuntimeError exception if there are any errors.

Parameter `ad_list` is a list of ClassAds containing job descriptions; typically, this is the list filled by the `ad_results` argument of the `submit` method call.

`retrieve( job_spec )`

Retrieve the output sandbox from one or more jobs.

Parameter `job_spec` is an expression string matching the list of job output sandboxes to retrieve.

`refreshGSIProxy(cluster, proc, filename, lifetime)`

Refresh the GSI proxy of a job with job identifier given by parameters `cluster` and `proc`. This will refresh the remote proxy with the contents of the file identified by parameter `filename`.

Parameter `lifetime` indicates the desired lifetime (in seconds) of the delegated proxy. A value of 0 specifies to not shorten the proxy lifetime. A value of -1 specifies to use the value of configuration variable `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`. Note that, depending on the lifetime of the proxy in `filename`, the resulting lifetime may be shorter than the desired lifetime.

`negotiate( (str)accounting_name )`

Begin a negotiation cycle with the remote schedd. The `accounting_name` parameter determines which user we will start negotiating with.

The returned object, of type `ScheddNegotiate` is iterable; its iterator will yield resource request ClassAds from the schedd. Each resource request represents a set of jobs that are next in queue for the schedd for this user.

The `ScheddNegotiate` additionally serves as a context manager, automatically destroying the negotiation session when the context is left.

Finally, `ScheddNegotiate` has a `sendClaim` method for sending claims back to the remote schedd based on a given resource request.

The `Submit` class:

`__init__( (dict)input = None )`

Create an instance of the `Submit` class.

Optional parameter `input` is a Python dictionary containing submit file key = value pairs. If omitted, the submit class is initially empty.

`expand( (str)attr )`

Expand all macros for the given attribute.

Parameter `attr` is the name of the relevant attribute.

Returns a string containing the value of the given attribute with all macros expanded.

`queue( (object)txn, (int)count = 1, (object)ad_results = None )`

Submit the current object to a remote queue. Parameter `txn` is an active transaction object (see `Schedd.transaction()`).

Optional parameter `count` is the number of procs to create (defaults to 1 if not specified).

Optional parameter `ad_results` is an object to receive the ClassAd resulting from this submit.

Returns the ClusterID of the submitted job(s).

Throws a RuntimeError if the submission fails.

`get( (str)attr, (str)default = None )`

Gets the value of the specified attribute.

Parameter `attr` is the name of the relevant attribute.

Optional parameter `default` is a default value to be returned if the attribute is not defined.

Returns a string containing the value of the attribute.

`setdefault( (str)attr, (str)default)`

Set a default value for an attribute.

Parameter `attr` is the name of the relevant attribute.

Parameter `default` is the value to which to set the given attribute if that attribute has not already been set.

Returns a string containing the value of the attribute.

`update( (object)submit )`

Copy the contents of a given Submit object into the current object.

Parameter `submit` is the Submit object to copy.

The `Collector` class:

`__init__( pool = None )`

Create an instance of the `Collector` class.

Optional parameter `pool` is a string with host:port pair specified or a list of pairs. If omitted, the value of configuration variable `COLLECTOR_HOST` is used.

`locate( (DaemonTypes)daemon_type, (str)name )`

Query the *condor_collector* for a particular daemon. Returns the ClassAd of the requested daemon.

Parameter `daemon_type` is the type of daemon; must be of the enum `DaemonTypes`.

Optional parameter `name` is the name of daemon to locate. If not specified, it searches for the local daemon.

`locateAll( (DaemonTypes)daemon_type )`

Query the *condor_collector* daemon for all ClassAds of a particular type. Returns a list of matching ClassAds.

Parameter `daemon_type` is the type of daemon; must be of the enum `DaemonTypes`.

`query( (AdTypes)ad_type, constraint=True, attrs=[], (str)statistics = '' )`

Query the contents of a *condor_collector* daemon. Returns a list of ClassAds that match the `constraint` parameter.

Optional parameter `ad_type` is the type of ClassAd to return, where the types are from the enum `AdTypes`. If not specified, the type will be `ANY_AD`.

Optional parameter `constraint` is a constraint for the ClassAd query. It defaults to `True`.

Optional parameter `attrs` is a list of attributes. If specified, the returned ClassAds will be projected along these attributes.

Optional parameter `statistics` is a list of statistics attributes to include, if they exist for the specified daemon.

`advertise( ad_list, command=UPDATE_AD_GENERIC, use_tcp = True )`

Advertise a list of ClassAds into the *condor_collector*.

Parameter `ad_list` is the list of ClassAds to advertise.

Optional parameter `command` is a command for the *condor_collector*. It defaults to `UPDATE_AD_GENERIC`. Other commands, such as `UPDATE_STARTD_AD`, may require reduced authorization levels.

Optional parameter `use_tcp` causes updates to be sent via TCP. Defaults to `True`.

`directQuery( (Collector)arg1, (DaemonTypes)daemon_type, (str)name = '', (list)projection = [], (str)statistics = '' )`

Query the specified daemon directly, instead of using the ClassAd from the *condor_collector* daemon. Returns the ClassAd of the specified daemon, after obtaining it from the daemon.

Parameter `arg1` is the *condor_collector* that will identify where to find the specified daemon.

Parameter `daemon_type` specified a daemon with an enum from `DaemonTypes`.

Optional parameter `name` specifies the daemon's name. If not specified, the local daemon is used.

Optional parameter `projection` is a list of attributes requested, to obtain only a subset of the attributes from the ClassAd.

Optional parameter `statistics` is a list of statistics attributes to include, if they exist for the specified daemon.

The `Negotiator` class:

`__init__( (ClassAd)ad = None )`

Create an instance of the `Negotiator` class.

Optional parameter `ad` is a ClassAd containing the location of the *condor_negotiator* daemon. If omitted, uses the local pool.

`deleteUser( (str)user )`

Delete a user from the accounting.

`user` is a fully-qualified user name, `"USER@DOMAIN"`.

`getPriorities( [(bool)rollup = False ] )`

Retrieve the pool accounting information. Returns a list of accounting ClassAds.

Optional parameter `rollup` identifies if accounting information, as applied to hierarchical group quotas, should be summed for groups and subgroups (`True`) or not (`False`, the default).

`getResourceUsage( (str)user )`

Get the resource usage for a specified user. Returns a list of ClassAd attributes.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`.

`resetAllUsage( )`

Reset all usage accounting.

`resetUsage( (str)user )`

Reset all usage accounting of the specified `user`.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`; resets the usage of only this user.

`setBeginUsage( (str)user, (time_t)value )`

Initialize the time that a user begins using the pool.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `value` is the time of initial usage.

`setLastUsage( (str)user, (time_t)value )`

Set the time that a user last began using the pool.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `value` is the time of last usage.

`setFactor( (str)user, (float)factor )`

Set the priority factor of a specified user.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `factor` is the priority factor to be set for the user; must be greater than or equal to 1.0.

`setPriority( (str)user, (float)prio )`

Set the real priority of a specified user.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `prio` is the priority to be set for the user; must be greater than 0.0.

`setUsage( (str)user, (float)usage )`

Set the accumulated usage of a specified user.

Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `usage` is the usage to be set for the user.

The `Startd` class:

`__init__( (ClassAd)ad = None )`

Create an instance of the `Startd` class.

Optional parameter `ad` is a ClassAd containing the location of the *condor_startd* daemon. If omitted, uses the local startd.

`drainJobs( (int)drain_type, (bool)resume_on_completion, (expr)check )`

Begin draining jobs from the startd. Returns a draining request_id.

Parameter `drain_type` type of drain to perform, from the DrainTypes enum either Fast, Graceful or Quick. Parameter `resume_on_completion` is true if the startd should start accepting jobs again once draining is complete, false if it should remain in the drained state. Parameter `constraint` An optional check expression which must be true on all slots for draining to begin.

`cancelDrainJobs( (int)request_id )`

Cancel a draining request.

Parameter `request_id` If specified, cancels only the drain command that returned the given request_id

The `SecMan` class accesses the internal security object. This class allows access to the security layer of HTCondor.

Currently, this is limited to resetting security sessions and doing test authorizations against remote daemons.

If a security session becomes invalid, for example, because the remote daemon restarts, reuses the same port, and the client continues to use the session, then all future commands will fail with strange connection errors. This is the only mechanism to invalidate in-memory sessions.

`__init__( )`

Create a `SecMan` object.

`invalidateAllSessions( )`

Invalidate all security sessions. Any future connections to a daemon will cause a new security session to be created.

`ping ( (ClassAd)ad, (str)command )`

or

`ping ( (string)sinful, (str)command )`

Perform a test authorization against a remote daemon for a given command.

Returns the ClassAd of the security session.

Parameter `ad` is the ClassAd of the daemon as returned by `Collector.locate`; alternately, the sinful string can be given directly as the first parameter.

Optional parameter `command` is the DaemonCore command to try; if not given, `DC_NOP` will be used.

The `Param` class provides a dictionary-like interface to the current configuration.

The `Param` class:

`__getitem__( (str)attr )`

Returns the configuration for variable `attr` as an object.

`__setitem__( (str)attr, (str)value )`

Sets the configuration variable `attr` to the `value`.

`__contains__( (str)attr )`

Determines whether the configuration contains a setting for configuration variable `attr`.

Returns `true` if the configuration does contain a setting for `attr`, and it returns false otherwise.

Parameter `attr` is the name of the configuration variable.

`__iter__( )`

Description not yet written.

`__len__( )`

Returns the number of items in the configuration.

`setdefault( (str)attr, (str)value )`

Behaves like the corresponding Python dictionary method. If `attr` is not set in the configuration, it sets `attr` to `value` in the configuration. Returns the `value` as an object.

`get( )` `get` description not yet written.
`keys( )` Return a list of configuration variable names that are defined in the configuration files.
`items( )` Returns an iterator of tuples. Each item returned by the iterator is a tuple representing a pair (attribute,value) in the configuration.
`update( source )` Behaves like the corresponding Python dictionary method. Updates the current configuration to match the one in object `source`.

The `RemoteParam` class provides a dictionary-like interface to the configuration of daemons.

The `RemoteParam` class:

`__getitem__( (str)attr )` Returns the configuration for variable `attr` as an object.
`__setitem__( (str)attr, (str)value )` Sets the configuration variable `attr` to the `value`.
`__contains__( (str)attr )` Determines whether the configuration contains a setting for configuration variable `attr`. Returns `true` if the configuration does contain a setting for `attr`, and it returns false otherwise. Parameter `attr` is the name of the configuration variable.
`__iter__( )` Description not yet written.
`__len__( )` Returns the number of items in the configuration.
`__delitem__( (str)attr )` If the configuration variable specified by `attr` is in the configuration, set its value to the null string. Parameter `attr` is the name of the configuration variable to change.
`setdefault( (str)attr, (str)value )` Behaves like the corresponding Python dictionary method. If `attr` is not set in the configuration, it sets `attr` to `value` in the configuration. Returns the `value` as an object.
`get( )` `get` description not yet written.
`keys( )` Return a list of configuration variable names that are defined for the daemon.
`items( )` Returns an iterator of tuples. Each item returned by the iterator is a tuple representing a pair (attribute,value) in the configuration.
`update( source )` Behaves like the corresponding Python dictionary method. Updates the current configuration to match the one in object `source`.

`refresh( )` Rebuilds the dictionary corresponding to the current configuration of the daemon.

The `Claim` class provides access to HTCondor's Compute-On-Demand facilities.

The `Claim` class:

`__init__( classad )` Create a Claim object. The `classad` argument provides a ClassAd describing the startd to claim.
`requestCOD( constraint, lease_duration )` Request a claim from the *condor_startd* represented by this object. The `constraint` specifies which slot in the startd to claim (defaults to 'true', which will result in the first slot becoming claimed). The `lease_duration` indicates how long the claim should be valid for. On success, the `Claim` object will represent a valid claim on the remote startd.
`release( (VacateTypes)vacate_type )` Release a *condor_startd* from this claim and shut down any running job. The `vacate_type` argument indicates the type of vacate to perform (Fast or Graceful); must be from VacateTypes enum.
`activate( (ClassAd)ad )` Activate a claim using a given job ad. The `ad` must describe a job to run.
`suspend()` Suspend an activated claim.
`renew()` Renew the lease on an existing claim.
`resume()` Resume a temporarily suspended claim.
`deactivate()` Deactivate a claim; shuts down the currently-running job, but holds onto the claim for future use.
`delegateGSIProxy()` Send an x509 proxy credential to an activated claim.

Module enums:

`AdTypes` A list of types used as values for the `MyType` ClassAd attribute. These types are only used by the HTCondor system, not the ClassAd language. Typically, these specify different kinds of daemons.
`DaemonCommands` A list of commands which can be sent to a remote daemon.
`DaemonTypes` A list of types of known HTCondor daemons.

`JobAction` A list of actions that can be performed on a job in a *condor_schedd*.
`SubsystemType` Distinguishes subsystems within HTCondor. Values may be `Master`, `Collector`, `Negotiator`, `Schedd`, `Shadow`, `Startd`, `Starter`, `GAHP`, `Dagman`, `SharedPort`, `Daemon`, `Tool`, `Submit`, or `Job`.
`LogLevel` The level at which events are logged. Values may be `Always`, `Error`, `Status`, `Job`, `Machine`, `Config`, `Protocol`, `Priv`, `DaemonCore`, `Security`, `Network`, `Hostname`, `Audit`, `Terse`, `Verbose`, `FullDebug`, `SubSecond`, `Timestamp`, `PID`, or `NoHeader`.

6.7.2 Sample Code using the `htcondor` Python Module

This sample code illustrates interactions with the `htcondor` Python Module.

```
$ python
Python 2.6.6 (r266:84292, Jun 18 2012, 09:57:52)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import htcondor
>>> import classad
>>> coll = htcondor.Collector("red-condor.unl.edu")
>>> results = coll.query(htcondor.AdTypes.Startd, "true", ["Name"])
>>> len(results)
3812
>>> results[0]
[ Name = "slot1@red-d20n35"; MyType = "Machine"; TargetType = "Job"; CurrentTime = time() ]
>>> scheddAd = coll.locate(htcondor.DaemonTypes.Schedd, "red-gw1.unl.edu")
>>> scheddAd["ScheddIpAddr"]
'<129.93.239.132:53020>'
>>> schedd = htcondor.Schedd(scheddAd)
>>> results = schedd.query('Owner =?= "cmsprod088"', ["ClusterId", "ProcId"])
>>> len(results)
63
>>> results[0]
[ MyType = "Job"; TargetType = "Machine"; ServerTime = 1356722353; ClusterId = 674143; ProcId = 0; CurrentTime = time() ]
>>> htcondor.param["COLLECTOR_HOST"]
'hcc-briantest.unl.edu'
>>> schedd = htcondor.Schedd() # Defaults to the local schedd.
>>> results = schedd.query()
>>> results[0]["RequestMemory"]
ifthenelse(MemoryUsage isnt undefined,MemoryUsage,( ImageSize + 1023 ) / 1024)
>>> results[0]["RequestMemory"].eval()
1L
>>> ad=classad.parse(open("test.submit.ad"))
>>> print schedd.submit(ad, 2) # Submits two jobs in the cluster; edit test.submit.ad to preference.
110
>>> print schedd.act(htcondor.JobAction.Remove, ["111.0", "110.0"])'
    [
        TotalNotFound = 0;
        TotalPermissionDenied = 0;
        TotalAlreadyDone = 0;
        TotalJobAds = 2;
        TotalSuccess = 2;
        TotalChangedAds = 1;
        TotalBadStatus = 0;
        TotalError = 0
    ]
>>> print schedd.act(htcondor.JobAction.Hold, "Owner =?= \"bbockelm\"")'
    [
        TotalNotFound = 0;
        TotalPermissionDenied = 0;
        TotalAlreadyDone = 0;
        TotalJobAds = 2;
        TotalSuccess = 2;
```

```
        TotalChangedAds = 1;
        TotalBadStatus = 0;
        TotalError = 0
    ]
>>> schedd.edit('Owner =?= "bbockelm"', "Foo", classad.ExprTree('"baz"'))
>>> schedd.edit(["110.0"], "Foo", '"bar"')
>>> coll = htcondor.Collector()
>>> master_ad = coll.locate(htcondor.DaemonTypes.Master)
>>> htcondor.send_command(master_ad, htcondor.DaemonCommands.Reconfig) # Reconfigures the local master and all children
>>> htcondor.version()
'$CondorVersion: 7.9.4 Jan 02 2013 PRE-RELEASE-UWCS $'
>>> htcondor.platform()
'$CondorPlatform: X86_64-ScientificLinux_6.3 $'
```

The bindings can use a dictionary where a ClassAd is expected. Here is an example that uses the ClassAd:

```
htcondor.Schedd().submit(classad.ClassAd({"Cmd": "/bin/echo"}))
```

This same example, using a dictionary instead of constructing a ClassAd:

```
htcondor.Schedd().submit({"Cmd": "/bin/echo"})
```

6.7.3 ClassAd Module

The `classad` module class provides a dictionary-like mechanism for interacting with the ClassAd language. `classad` objects implement the iterator interface to iterate through the `classad`'s attributes. The constructor can take a dictionary, and the object can take lists, dictionaries, and ClassAds as values.

`classad` module functions:

`parseOne( input, parser=Auto )` Parse the entire `input` into a single ClassAd. In the presence of multiple ClassAds or blank lines, continue to merge ClassAds together until the entire string is consumed. Returns a `classad` object. Parameter `input` is a string-like object or a file pointer. Parameter `parser` specifies which ClassAd parser to use.
`parseNext( input, parser=Auto )` Parse the next ClassAd in the input string. Advances the `input` object to point after the consumed ClassAd. Returns a `classad` object. Parameter `input` is a file-like object. Parameter `parser` specifies which ClassAd parser to use.
`parse( input )` *This method is no longer used*. Parse input into a ClassAd. Returns a ClassAd object. Parameter `input` is a string-like object or a file pointer.
`parseOld( input )` *This method is no longer used*. Parse old ClassAd format input into a ClassAd. Returns a ClassAd object. Parameter `input` is a string-like object or a file pointer.
`version( )` Return the version of the linked ClassAd library.
`lastError( )` Return the string representation of the last error to occur in the ClassAd library.

`Attribute( name )`

Given the string `name`, return an `ExprTree` object which is a reference to an attribute of that name. The ClassAd expression `foo == 1` can be constructed by the python `Attribute("foo") == 1`.

`Function( name, arg1, arg2, ... )`

Given function name `name`, and zero-or-more arguments, construct an `ExprTree` which is a function call expression. The function is not evaluated. The ClassAd expression `strcat("hello ", "world")` can be constructed by the python `Function("strcat", "hello ", "world")`.

`Literal( obj )`

Given python object `obj`, convert it to a ClassAd literal. Python strings, floats, integers, and booleans have equivalent literals.

`register( function, name=None )`

Given the python function `function`, register it as a ClassAd function. This allows the invocation of the python function from within a ClassAd evaluation context. The optional parameter, `name`, provides an alternate name for the function within the ClassAd library.

`registerLibrary( path )`

Given a file system `path`, attempt to load it as a shared library of ClassAd functions. See the documentation for configuration variable `CLASSAD_USER_LIBS` for more information about loadable libraries for ClassAd functions.

Standard Python object methods for the `ClassAd` class:

`__init__( str )`

Create a ClassAd object from string, `str`, passed as a parameter. The string must be formatted in the new ClassAd format.

`__len__( )`

Returns the number of attributes in the ClassAd; allows `len(object)` semantics for ClassAds.

`__str__( )`

Converts the ClassAd to a string and returns the string; the formatting style is new ClassAd, with square brackets and semicolons. For example, `[ Foo = "bar"; ]` may be returned.

The `classad` object has the following dictionary-like methods:

`items( )`

Returns an iterator of tuples. Each item returned by the iterator is a tuple representing a pair (attribute,value) in the ClassAd object.

`values( )`

Returns an iterator of objects. Each item returned by the iterator is a value in the ClassAd.

If the value is a literal, it will be cast to a native Python object, so a ClassAd string will be returned as a Python string.

`keys( )`

Returns an iterator of strings. Each item returned by the iterator is an attribute string in the ClassAd.

`get( attr, value )` Behaves like the corresponding Python dictionary method. Given the `attr` as key, returns either the value of that key, or if the key is not in the object, returns `None` or the optional second parameter when specified.
`__getitem__( attr )` Returns (as an object) the value corresponding to the attribute `attr` passed as a parameter. ClassAd values will be returned as Python objects; ClassAd expressions will be returned as `ExprTree` objects.
`__setitem__( attr, value )` Sets the ClassAd attribute `attr` to the `value`. ClassAd values will be returned as Python objects; ClassAd expressions will be returned as `ExprTree` objects.
`setdefault( attr, value )` Behaves like the corresponding Python dictionary method. If called with an attribute, `attr`, that is not set, it will set the attribute to the specified `value`. It returns the value of the attribute. If called with an attribute that is already set, it does not change the object.
`update( object )` Behaves like the corresponding Python dictionary method. Updates the ClassAd with the key/value pairs of the given object. Returns nothing.

Additional methods:

`eval( attr )` Evaluate the value given a ClassAd attribute `attr`. Throws `ValueError` if unable to evaluate the object. Returns the Python object corresponding to the evaluated ClassAd attribute.
`lookup( attr )` Look up the `ExprTree` object associated with attribute `attr`. No attempt will be made to convert to a Python object. Returns an `ExprTree` object.
`printOld( )` Print the ClassAd in the old ClassAd format. Returns a string.
`quote( str )` Converts the Python string, `str`, into a ClassAd string literal. Returns the string literal.
`unquote( str )` Converts the Python string, `str`, escaped as a ClassAd string back to a Python string. Returns the Python string.
`parseAds( input, parser=Auto )` Given `input` of a string or file, return an iterator of ClassAds. Parameter `parser` tells which ClassAd parser to use. Note that automatic selection of ClassAd parser does not work on stream input. Returns an iterator.

`parseOldAds( input )` *This method is no longer used*. Given `input` of a string or file, return an iterator of ClassAds where the ClassAds are in the Old ClassAd format. Returns an iterator.
`flatten( expression )` Given `ExprTree` object `expression`, perform a partial evaluation. All the attributes in `expression` and defined in this object are evaluated and expanded. Any constant expressions, such as `1 + 2`, are evaluated. Returns a new `ExprTree` object.
`matches( ad )` Given `ClassAd` object `ad`, check to see if this object matches the `Requirements` attribute of `ad`. Returns `true` if it does.
`symmetricMatch( ad )` Returns `true` if the given `ad` matches this and this matches `ad`. Equivalent to `self.matches(ad) and ad.matches(self)`.
`externalRefs( expr )` Returns a python list of external references found in `expr`. In this context, an external reference is any attribute in the expression which is *not* found in the `ClassAd`.
`internalRefs( expr )` Returns a python list of internal references found in `expr`. In this context, an internal reference is any attribute in the expression which is found in the `ClassAd`.

The `ExprTree` class object represents an expression in the ClassAd language. The python operators for `ExprTree` have been overloaded so, if `e1` and `e2` are `ExprTree` objects, then `e1 + e2` is also a `ExprTree` object. Lazy-evaluation is used, so an expression `"foo" + 1` does not produce an error until it is evaluated with a call to `bool()` or the `.eval()` class member.

`ExprTree` class methods:

`__init__( str )` Parse the string `str` to create an `ExprTree`.
`__str__( )` Represent and return the ClassAd expression as a string.
`__int__( )` Converts expression to an integer (evaluating as necessary).
`__float__( )` Converts expression to a float (evaluating as necessary).
`eval( )` Evaluate the expression and return as a ClassAd value, typically a Python object.

Module enums:

`Parser` Tells which ClassAd parser to use. Values may be `Auto`, `Old`, or `New`.

6.7.4 Sample Code using the `classad` Module

This sample Python code illustrates interactions with the `classad` module.

```
$ python
Python 2.6.6 (r266:84292, Jun 18 2012, 09:57:52)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import classad
>>> ad = classad.ClassAd()
>>> expr = classad.ExprTree("2+2")
>>> ad["foo"] = expr
>>> print ad["foo"].eval()
4
>>> ad["bar"] = 2.1
>>> ad["baz"] = classad.ExprTree("time() + 4")
>>> print list(ad)
['bar', 'foo', 'baz']
>>> print dict(ad.items())
{'baz': time() + 4, 'foo': 2 + 2, 'bar': 2.100000000000000E+00}
>>> print ad
    [
        bar = 2.100000000000000E+00;
        foo = 2 + 2;
        baz = time() + 4
    ]
>>> ad2=classad.parseOne(open("test_ad", "r"));
>>> ad2["error"] = classad.Value.Error
>>> ad2["undefined"] = classad.Value.Undefined
>>> print ad2
    [
        error = error;
        bar = 2.100000000000000E+00;
        foo = 2 + 2;
        undefined = undefined;
        baz = time() + 4
    ]
>>> ad2["undefined"]
classad.Value.Undefined
```

Here is an example that illustrates the dictionary properties of the constructor.

```
>>> classad.ClassAd({"foo": "bar"})
[ foo = "bar" ]
>>> ad = classad.ClassAd({"foo": [1, 2, 3]})
>>> ad
[ foo = { 1,2,3 } ]
>>> ad["foo"][2]
3L
>>> ad = classad.ClassAd({"foo": {"bar": 1}})
>>> ad
[ foo = [ bar = 1 ] ]
>>> ad["foo"]["bar"]
1L
```

Here are examples that illustrate the `get` method.

```
>>> ad = classad.ClassAd({"foo": "bar"})
>>> ad
[ foo = "bar" ]
>>> ad["foo"]
'bar'
>>> ad.get("foo")
'bar'
>>> ad.get("foo", 2)
'bar'
>>> ad.get("baz", 2)
2
>>> ad.get("baz")
>>>
```

Here are examples that illustrate the `setdefault` method.

```
>>> ad = classad.ClassAd()
>>> ad
[  ]
```

```
>>> ad["foo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'foo'
>>> ad.setdefault("foo", 1)
1
>>> ad
[ foo = 1 ]
>>> ad.setdefault("foo", 2)
1L
>>> ad
[ foo = 1 ]
```

Here is an example that illustrates the use of the iterator `parseAds` method on a history log.

```
>>> import classad
>>> import os
>>> fd = os.popen("condor_history -l -match 4")
>>> ads = classad.parseAds(fd, classad.Parser.Old)
>>> print [ad["ClusterId"] for ad in ads]
[23389L, 23388L, 23386L, 23387L]
>>>
```

Next Up Previous Contents Index

Next: 7. Platform-Specific Information **Up:** 6. Application Programming Interfaces **Previous:** 6.6 The HTCondor Perl **Contents** **Index**