ATS Documentation

Release 5.5

Paul F. Dubois and Nu Ai Tang

Mar 14, 2024

CONTENTS

CHAPTER

ONE

PURPOSE AND FEATURES

The Automated Testing System (ATS) is an open-source, Python-based tool for automating the running of tests of an application. ATS can test any program that can signal success or failure via its exit status.

ATS is distributed, introspective, and scalable.

- It is distributed in two senses. First, there is no central database of tests to run. Tests may be spread over many directories and usually adding a test in a subdirectory is entirely a local operation.
- Introspective means that a test can be runnable by someone who is not an expert, yet runnable with different arguments by someone who is. An application test may contain within itself, in comments, directions for how to run itself in one or more ways. An expert may run these tests normally using his application; but when ATS runs it, it runs the application according to the special comments within the input.
- Depending on the available resources, the execution of the tests can be done over many processors and hosts, in parallel. Distributed execution and test specification helps ATS stay scalable.

Other features of ATS include:

- A test may depend on another test, and will not be executed unless their parent test succeeds.
- Tests may be filtered out (that is, not executed) in many ways. These may include number of processors, time limit, platform, or other user-defined criteria.
- A level may be given to each test, and used to stratify a test suite into subsets of increasing thoroughness.
- ATS is extensible. The ats driver script does almost nothing except import the ats module and call the ats.manager.main() method. It may suit your purposes to make a different driver that does things before or after this invocation. The ats script uses the assets of the module ats to provide a command line interface-type testing system. Other interfaces, such as a GUI interface, are possible.
- ATS makes it easy in particular to postprocess the results of the testing by registering routines to be executed after the tests have completed, but before exiting.
- A facility is provided to make it easy to port ATS to new machines such as parallel processors and multi-noded distributed machines, or to take advantage of multiple cores. The 'stock' ATS will run up to two tests at once, each of them standard serial jobs (np = 1 in what follows).

While ATS input can be written using the full power of Python, the basic operations require only a few statements written in a special vocabulary that is not be hard to learn. For example:

```
test(executable="/my/path/to/my/code",
     clas="-input mydeck delta=3",
     np=3)
```

executes the given executable with the given command-line arguments (clas), launching the job is parallel on 3 processors.

Note: A note on function signatures While this document assumes you can learn the basics of Python on your own, function signatures require careful understanding. In Python, the definition of a function parameter can have one or two asterisks in front of a name.

- When calling such a function, in the place of a parameter with one asterisk in front of it, you can give zero or more comma-separated values as a value, which the function will receive as a list.
- When a parameter name has two asterisks in front of it in the function definition, you can give zero or more comma-separated keyword = value pairs when you call it, which the function will receive as a dictionary.

For example, the ATS `source` function has the signature:

```
source(*paths, **vocabulary)
```

which means any of the following are legitimate calls to it:

```
source('foo.py')
source('foo.py', 'goo.py')
source('foo.py', physics="on", music = "off")
source(physics="on", music = "off")
source()
```

As it happens, the last of these doesn't do anything, but it is a legitimate call.

Note: All of the *paths* arguments must come before the first of the *vocabulary* arguments.

1.1 Download and Install

Installation of ATS is easy. Unpack the distribution and in the top-level directory execute:

```
python setup.py install
```

Public releases are at http://code.google.com/p/ats

The README.txt file contains installation instructions. ATS has been tested with Python 2.6 or later, available at http://python.org.

ATS should translate to Python 3 by using the 2to3 utility but this has not yet been tried.

ATS should work, or be made to work, on any system which can run Python via a command window. In particular it works out of the box on any Linux or Mac system. ATS works on Windows but experience there is limited.

1.2 History

ATS was written by Paul F. Dubois at Lawrence Livermore National Laboratory, (LLNL) in about 2003. Although an open-source release was made, the software was highly oriented to the LLNL computer systems and one particular simulation, ATS has been in continuous use since then.

A revision in 2010-11 has compartmentalized the LLNL-specific system details, and we have added new features to make the software more generally applicable and more easily portable.

The support team at LLNL includes Nu Ai Tang, T. J. Alumbaugh, and Ines Heinz. You can contact the author at dubois1@llnl.gov. For help with the LLNL features contact tang10@llnl.gov.

ATS was written to test scientific simulations, although it can be used for any program that can be run with a command-line, does not require interaction, and which can signal its own success or failure via its exit status (or be executed via a shell program with those properties).

In general scientific programs do not produce predictable printed output, and so comparison of output files, so common in the testing literature, is not normally useful.They also are generally long-running and resource-consuming; hence ATS emphasises filtering, parallel execution, and prioritization under user control. Provision for supporting batch execution is also provided.

1.3 LLNL Notes

The LC distribution includes an LC directory containing definitions for the local machines and the batch system. To make use of the features of LC machines you will need to set either SYS_TYPE or MACHINE_TYPE. To install the LC machines, run:

```
python setup.py install
```

in the LC directory after you have done so in the main ATS directory.

For help join the mailing list ats@lists.llnl.gov.

1.4 About The Documentation

This document is licensed under the terms of the LICENSE.txt file in the ATS distribution.

This documentation is written in reStructuredText, the standard language used by the Python documentation project. You should find the source, available in the distribution, readable even without rendering. It can be if desired rendered into plain text files, web pages, PDF files, and other formats using the tools of the *Sphinx* project. The source files are located in the `source` subdirectory of the `docs` directory. The Makefile in the `docs` directory will render the documents into the `build` subdirectory if appropriate parts of Sphinx have been installed.

If you install `setuptools` into your Python, you can get Sphinx with:

```
easy_install -U Sphinx
```

1.4.1 The ATS Tutorial, Andyroid

Introduction To Andyroid

We have seen a brief overview of ATS and its rationale. This part of the document discusses an extended example and some hints about how to use ATS. It is followed by a reference section. Few adventurers survive the reference section; take an easy trip with this tutorial before venturing into that jungle.

The sources you need to follow along interactively are in the `Examples/Andyroid` subdirectory of the ATS distribution. However, you can also just read along.

The premise of this tutorial is that we have a program called *andyroid*; it has a post-processor named *andyroidPoster* that runs using the output file of *andyroid* as input.

These two executables are located in subdirectory *andyroid*. To avoid having to compile anything for this example, these are in fact scripts, so that to execute *andyroid* is really accomplished by executing:

```
python andyroid/andyroid.py
```

but you can just imagine that andyroid is a compiled program that has been installed in that subdirectory. You can see the options *andyroid* has by executing `python andyroid/andyroid.py --help`.

-h, --help show this help message and exit

-i FILE, --input=FILE input file name

-o FILE, --output=FILE output file name

--delta Add delta?

--alpha=ALPHA A vital parameter

From here on out we will assume *andyroid* is an alias for `python andyroid/andyroid.py` so that we don't add to the confusion with the extra text that would not be present for a real program. Just pretend *andyroid* is a compiled program.

Running Andyroid under ATS

Let's suppose that we normally test our program *andyroid* by using this command:

```
andyroid -i test1.in -o test1.out
```

How do we get ATS to execute this test for us? The simplest way is to make a file `simpleTestSuite.ats` that contains:

```
import os
# this is the system-independent way to say "python andyroid/andyroid.py"
andyroid =sys.executable + ' ' + os.path.join("andyroid", "andyroid.py")

test(executable = andyroid,
     clas = '-i test1.in -o test1.out', label= 'test1')
```

(`clas` stands for "Command Line ArgumentS"). Then execute:

ats simpleTestSuite

Note: this is not the recommended strategy, but there is a lot to learn by starting simply. ATS has lots of command-line options as detailed in the reference part of this document, but we don't need any yet.

Go ahead and try it. You'll notice some interesting things here.

- The input file simpleTestSuite was found even though we didn't put on the .ats extension. ATS tries the name, then the name with .ats added, and finally the name with .py added.
- We constructed the name of the executable using sys.executable so that it would use the same Python that ats is being run with. In general, ATS tests the executable to be sure it exists and does not rely on your path to find it, so you must be precise. This is to avoid having all your tests pass when in fact you didn't execute the program you intended to test.
- `label` is a name for this test. A test has both a unique serial number, and a unique label. (If the label isn't unique ATS will make it unique after all the tests have been collected from your input file(s).)
- The output from the program we test (and separately, its standard error), and the output from ats itself, are put in a single directory. This directory has a name that contains the kind of computer we're running on, and the time. The directory has the extension `.logs`.

  The output files for a given test contain the test's serial number, a simplified form of the label, and the time. That doesn't mean all the output from the test goes there; if the test creates files it creates them in the directory where ATS is running them, which is by default the directory of the "sourced" file that specified the test. That doesn't have to be so, as explained in the reference section entry for the *test function*.

  So, when you ran `simpleTestSuite`, *andyroid* created a `something.logs` directory, containing `ats.log`. Unless the test failed, the standard output and standard error (which were in that same directory) have been deleted.

  To make ATS keep the output we can add an option to the test command, `keep = True`. Or, we can run ATS with a `--keep` option, which will keep the output of any test that doesn't have `keep = False` as an option.

You'll also notice the log has full information on which tests passed or failed, and has various summaries and the list of what tests were started in what order. Additional information on scheduling is available in `atss.log`, especially when using the `--verbose` or `--debug` options.

Also in the log directory after ATS has finished execution is a file named "atsr.py", where the "r" stands for "results". This file can be used in postprocessing; see *Results Facility* for details.

ATS Execution Phases

ATS works in six phases.

1. Read the files on the command line.
2. Examine the collection of tests, make sure every test has a distinct label, and identify batch, interactive, and ineligible jobs (for example, one that needs more processors than are available).
3. Dispatch any batch jobs that have been specified to the batch system. If there is no batch system, such jobs are usually skipped, but this can be overridden by the '–allInteractive' option.
4. Run the tests.
5. Report the results
6. Run any postprocessors the user has defined.

Here are the details.

Phase 1: Sourcing

The first phase is to read the files you specify on the command line in the order you gave them. This is called *sourcing* them, because it is equivalent to using ATS's `source` command.

A file being sourced is written in Python using some already built-in features, as we discuss later. In `simpleTestSuite`, we are able to refer to a function called `test`, which is already defined for us.

A test is created for each `test` or `testif` statement that is executed. However, the ATS statements such as `test` can be mixed with arbitrary Python statements.

The `test` or `testif` statements return a value, a test object. This object contains all the information about the test; its attributes are documented in the reference manual.

Warning: A test is not executed when the test function is executed.

The input language creates the illusion that the test function is causing the test itself to be executed. And it is ... eventually, but not now. The test statement creates a test object and puts it in a big list of test objects, but it doesn't execute any tests until it is entirely done sourcing files.

Consequently, you may not test the truth value of a test object:

```
if test(...):   # ERROR CANNOT DO THIS
   test(...)

t = test(...)
if t: ...        # NOR THIS
```

This coding is disallowed because it looks like it is making one test depend upon another, but it isn't. The tests are not executed at this stage but rather later.

To make one test depend on another's success, we do this:

```
t = test(...)
testif(t, ...)
```

As the tests are collected, the filters that have been defined so far are used to see if the test should be attempted or not. Besides any user-defined filters there are built-in filters on the number of processors, `np`, and the `level`. The level is simply an easier-to-use filter that lets us execute just a portion of a test suite.

Two more functions, `group` and `endgroup`, can be used to group together a set of tests that are to be considered as a unit for reporting success or failure, and optionally for protecting one or more directories from interference from other tests.

The `wait` function can be used to divide source files into portions, so that the tests defined in that source file, after a `wait()` call, execute only when the tests declared above it are completed.

All these features are described in more detail in the reference material chapter *Controlling Input*.

In the log, the end of the input phase is marked with a message that says, "Input complete."

Phase 2: Sorting

The tests are examined to determine these things:

- Is the test to be executed interactively or in a batch system?
- Does the test depend on another (via a `testif` statement) so that its execution must follow that of its parent (and be cancelled if the parent fails?).
- Is the test a member of a group, or subject to a `wait` statement?
- Are there sufficient CPU resources to run the test?

From all this information, the list of tests that each test must wait for is calculated, and a priority is assigned.

Phase 3: Batch

Any tests that are scheduled for batch are sent off to be handled by the batch system. The details of how that is done and how you find out what happened depends on the particular batch system.

Phase 4: Execution

ATS must decide which tests to start given the available resources. To that end, each test has a priority. We can assign that priority (an integer) in the test statement itself, but if we do not, a priority is calculated that reflects the value of np (the number of processors the test requires) in the test and the priorities of any tests that must wait for this one to finish.

As a result, parent tests tend to be executed earlier so that they do not become a bottleneck. But, depending on the available resources, lower-priority jobs may be used to keep the machine "full".

You can see from the logs (especially atss.log``) which tests were started. If you see that a test ends up executing for a long time after all the others are finished, you can give it a higher priority. If you aren't getting the behavior you expect, see the reference chapters for further details, especially *directory blocking*.

After test execution is completed, a file named `continue.ats` is written into the logs directory if any of the tests failed. After fixing the problem, you can use `continue.ats` as an additional input file to another ats run. This allows you to fix as many problems as possible before attempting a full test suite again.

Phase 5: Report

Reports are made about the tests, followed by summaries. Some tests can be made to report on the terminal only if they fail, using the `record` or `group` options. These reports are made to the log. Information about tests that finished can be seen immediately by using the `--verbose` command-line option.

Phase 6: Postprocessing

Any functions registered by the user for post-processing are executed.

`onExit`(*f*)

onExit(f) can be called with a the name of a function that takes one argument, the ATS manager object. At the end of the ATS run this function will be called. The function f can do whatever it likes.

Multiple functions can be registered and they will be called in the order in which they were registered. Possible applications are printing reports, making graphs, etc.

Note: The master testlist is manager.testlist.

A file `atsr.py` is written into the log directory and can be used for postprocessing after ATS has finished. Using this facility, you can compare runs or analyze previous runs. See *Results Facility* in the reference section.

Postprocessing Example

Here we add coding to our ATS input file to print out information for those tests that were filtered out:

```
def showFiltered (manager):
   filtered = [t in manager.testlist if t.status is FILTERED]
   log("Detailed list of filtered tests.")
   log.indent()
   for t in filtered:
       log(t.serialNumber, t.name, t.note)
   log.dedent()
onExit(showFiltered)
```

It would also work to put the showFiltered function in a file showf.py, run:

```
python -i <logdirectory>/atsr.py showf.py

>>> showFiltered(state)
```

The file `atsr.py` defines a variable `state` that contains information equivalent to the `manager` object. Using the `-i` flag to Python, you can interactively examine the results of the ATS run.

Debugging Techniques

level = debug(ivalue = None)

ivalue is an integer, or omitted.

With no arguments, debug returns the current debug level; with an argument it sets the level.:

```
if debug():
    test(....)
old = debug()  #save current value
debug(1)
...  # debug is true in this section
debug(old)   # restore previous value
```

So you can have various levels of debugging in your own coding:

```
myDebugLevel = 2
dsave = debug()   # save the current value
debug(myDebugLevel)
if debug():
     .... do some stuff...
if debug() >= 2:
    ... do some more stuff...
debug(dsave)   # restore original value
```

Note: The `--debug` command-line option is equivalent to a `debug(1)` call at the start of your input.

Just remember you can't do an `if` on a test object, and it is rather pointless to do something right after a test statement because the test won't run until the input is all finished.

```
logDefinition(name1, ... , echo=True, logging=True)
```

can be used to log the named vocabulary words , or with no words, all the names.

Debugging Scheduling

If you run in debug or verbose mode, you will get a lot of information about what affected the job schedule by examining the `atss.log` file. Entries appear showing whether a job that could have executed has been blocked because it is waiting for directory blocking (B), waits or dependencies (W), or for adequate numbers of processors (C for "CPUs").

Structuring Your Test Suite

It is rare that a test suite of any size becomes an all-or-nothing affair. The more tests there are, the more the need to run selected sets for selected purposes. However, having the same test specifications repeated in a variety of input files for ATS is an invitation to maintenance headaches.

Simple First Steps

We could do the entire test suite by making `simpleTestSuite` larger, listing one test after another in a single file. As we will see later, various filters and level indicators can be used to make it possible to execute selected subsets of the test list.

For example (`inline.ats`):

```
import os, sys
codeDir = os.path.abspath(os.path.join(os.getcwd(), 'andyroid'))
andyroid = '%s %s/andyroid.py' % (sys.executable, codeDir)
andyroidPoster = '%s %s/andyroidPoster.py' % (sys.executable, codeDir)
stick(clas="-i %(inputFile)s -o %(outputFile)s %(opts)s")
stick(opts='')

glue(level=10)
test(executable=andyroid, inputFile='test1.in', outputFile="test1.out",
          label="test1")

glue(level=20)
t = test(executable=andyroid, inputFile='test1.in', outputFile="test1d.out",
          opts="--delta", label="test1d")
testif(t, clas = 'test1d.out', executable=andyroidPoster, label='test1dpost',
       keep=1)
```

However, in our experience a centralized test file is not a good idea except on a small project. If you have several developers who work in a distributed source tree, it is better to have the tests near where the developers work, so that they can add new tests and don't have to fight over a single file containing a master test list. Instead the master file, or a few files for different purposes, should contain mostly `source` statements to source the master files of various subdirectories; e.g.:

```
source('subdirectory1/area1.ats')
source('subdirectory2/area2.ats')
source('subdirectory3/area3.ats')
```

and so on down the tree until you get to files that actually specify tests in different areas.

However, when you start to do this, you do lose one thing. Remember our line that specified what "andyroid" meant? That would have to be repeated all the way down unless we do something about it. That's not so bad until the day you want to run some alternate version of andyroid. The solution is to define the symbol "andyroid" so that it will be known in any subsequent "sourced" files:

```
andyroid = '/my/path/to/andyroid'
define(andyroid=andyroid)
source('subdirectory1/area1.ats')
...
```

We will see this in action later.

Understanding The Test Statement

In Python, functions often take arguments of the form name = value. These are called keyword-value pairs.

The *test* function takes these arguments:

- Zero, one or two positional arguments, followed by
- An arbitrary number of keyword-value pairs. These keyword-value pairs are collectively called the "options".

The possible forms are:

```
test(script, clas, option1 = value1, ...)
test(script, option1 = value1, ...)
test(option1 = value1, ...)
```

The *testif* function is the same with an additional (required) first argument, the value returned by a previous *test* or *testif* function.

Understanding Test Options

Some options have default values. Here is a list of the arguments and options in approximate level of importance or likelyhood of use:

```
* script can be given by an option rather than as a positional
  argument, or omitted.

* clas can likewise be given as an option and in fact must be if
  script is omitted, or omitted.

* executable = 'path/to/executable' is the program to be tested.
  If not given, the executable is the one specified with the ``-e``
  (or ``--executable``) command-line option, which defaults to Python
  itself.

  Your executable may include options, such as '/path/to/executable -f',
```

(continues on next page)

(continued from previous page)

```
  or may be given as a list of components, such as
  ['/path/to/executable', '-f']. If the path contains a space, you
  must use the list form to avoid ambiguity.

* np = 0  is the number of processors required. Zero means 1 processor
          but may differ in consequence from np = 1 on some machines.

* label should always be specified to help you understand which test
  is referred to in ATS's output. It defaults to the script name.

* name is calculated from the name of the executable, but you can
  set it explicitly. The full name of the test is "name (label)".

* batch = False; if set to True, the job is executed in batch if possible
  and otherwise not at all unless ``--allInteractive`` is used.

* level = 1 is the level of the test, which is subject to the built-in
  level filter controlled by the ``--level`` command-line option.

* priority is calculated for you if not given.

* independent = False; if set to True, the test can be executed when
  CPU resources are available. If False, the test will not be able to execute
  until no other test is running in that directory. See also the
  :ref:`group facility <group_statement>`.

* timelimit has a default value of 30m, or as set on the command line
  with ``--timelimit``. The test will be killed and given a timed-out
  status if it is not finished running after this much time.

* keep = 0; if set to 1 (or set to 1 by the ``--keep`` option), the output
  files are kept even for tests that passed. If set to 2, the standard
  error file is also kept.

* check = False; if True, a test that passes is listed as one whose
  output needs to be checked by hand.

* record = True; record can be set to False to omit summary reports of
  this test unless it fails. You might do this with tests that are
  simply post-processing followups to a test upon which they depend.

* hideOutput = False; if set to true, any captured output is not
  printed in the log. (See the discussion Using Magic Output).

* directory defaults to the directory in which script resides; or if
  script is not given, to the directory in which the file being
  sourced statement resides. ATS will execute ``executable`` in this
  directory.

* magic = "ATS:"; if an output line starts with this symbol, the
  rest of the line is stored in test.output. The newline at the
  end is stripped off.
```

(continues on next page)

(continued from previous page)

```
* SYSTEMS if given is a list of machine names on which this test is to
  be executed. Otherwise the test will be executed if otherwise eligible.
```

Resolving Option Values

For each option keyword there is a final value determined as follows:

- Start with the default value, if any.
- Apply values that have been set using the `glue` function.
- Apply values that have been set using the `tack` function
- Apply values that have been set using the `stick` function.
- Apply values that have been set using the `group` function.
- Apply values in the test's options.

This final value is used. The dictionary of final values is used for interpolation into *script* and *clas*, and for filtering.

Here are the scopes of the various ways of setting values:

- Values set in a test statement apply only to that test.
- Values set with `stick` apply only to test statements that follow it within the same file. A file sourced by this one does not see the *stuck* value.
- Values set with `tack` apply until to all subsequent test statements until the file being currently sourced is completed. If this file sources another, the tacked value applies in it too.
- Values set with `glue` apply to all subsequent test statements until overwritten.
- Values set with `group` apply to all tests defined within the group. This scope also ends at the end of a source file.
- Values set on the command line apply for the entire run.

User-Defined Options

You can add any keyword-value pairs you want to the `test` statements, and set defaults for them with the `glue`, `tack`, or `stick` statements. Then you can use them for filtering or for interpolation into `clas` and `script`.

Interpolation of the options into clas and script is done by using Python's % operator. For example, if clas = "-in %(inputFile)s", and we have an option inputFile = 'test1.in', the result will be clas = "-in test1.in".

The user can define options for the purpose of controlling which tests get executed. For example, if you do this at the top of your input:

```
glue(threshold = 0.)
```

and in some tests you have a different value:

```
test(...)
test(..., threshold = 1., label = 'just me!')
stick(threshold=10.)
test(...)
test(...)
```

then you can execute ATS with a filter to screen out those tests where threshold is outside of some range:

```
ats mytest -f 'threshold >=0.5 and threshold <=2.0'
```

This would execute only the second test above.

Default values can also be defined locally with `glue`, `tack`, `stick` and `group` directives, and filtered with `filter` directives. Including a small file with such values and filters might be an effective way to define a suite:

```
ats mydefinitions mytest
```

where mydefinitions contains glue, tack, and filter specifications.

Pop quiz: in the preceding sentence, why isn't stick mentioned?

Understanding Defines

When a file is sourced, the language in which it is parsed consists of any Python statement or built-in function, plus a limited vocabulary that includes functions like `test`, `testif`, `glue`, `tack`, `stick`, and `log`. The user can manipulate this list for *subsequent* sourced files using these functions:

- define(name=value) adds the name with the given value to the vocabulary.
- undefine(name) removes name from the vocabulary.
- logDefinition(name) prints the value of name in the vocabulary; with no name given, it prints the entire vocabulary.
- get(name) retrieves the value associated with name.

If you source a file that adds to the vocabulary, it will not apply in the rest of the file that did the sourcing. For example:

```
source('mydefs.ats')  # in mydefs.ats, define(foo=value) is executed.
source('file2.ats')  # foo will be defined while sourcing file2.ats.
test(executable=foo)  # Error! foo not defined here
```

To remedy this we use the `get` function:

```
foo = get('foo')
test(executable=foo)  # foo defined here now.
```

Here's an important fact about sourcing: a file is never sourced twice. If it has already been sourced, it is skipped. That means that it is not expensive to do:

```
source('mydefs.ats')
foo = get('foo')
```

in any input files. It won't matter which of them is executed first, they will all get the definition for foo that they need.

Defining functions

Note that you can define anything to put it in the vocabulary, including Python functions. For example, suppose we wish to define a function that executes andyroid and its post-processor andyroidPoster and which has an interface of our choosing. Here is an example (file andyroid/andyroid.ats):

```
import os, sys
here = os.getcwd()
codeDir = os.path.abspath(os.path.join(here))
defaultAndyroid = '%s %s/andyroid.py' % (sys.executable, codeDir)
defaultAndyroidPoster = '%s %s/andyroidPoster.py' % (sys.executable, codeDir)

andyroid = os.environ.get('andyroid', defaultAndyroid)
andyroidPoster = os.environ.get('andyroidPoster', defaultAndyroidPoster)

count = 0
def runAndPost(inputFile, outputFile=None, label=None,
               delta = False,
               alpha = None, **options):
    global count
    count += 1
    if outputFile is None:
        outputFile = 'andyroid%05d.out' % count
    if label is None:
        label = inputFile
    clas = "-i %s -o %s" % (inputFile, outputFile)

    if delta:
        clas += " --delta"

    if alpha is not None:
        clas += " --alpha %f" % alpha

# Test the code
    t = test(clas=clas, executable=andyroid, label = label,
             name="Andyroid", **options)

# Test the postprocessor
# report = False means omit separate report for postprocessor if it passes.
    testif(t, clas=outputFile, executable=andyroidPoster, label=t.name,
             report=False, name="AndyroidPoster", keep = 1)
    return t

define(andyroid=runAndPost)
```

(We return the test `t` in case we later want to have access to it, such as making another test depend on it by defining a similar `runAndPostIf(t, ...)` function.)

Now we can define a new file `testSuite.ats`:

```
source('andyroid/andyroid.ats')
andyroid = get('andyroid')
andyroid('test1.in', label='test1')
andyroid('test1.in', label='test1d', delta=True)
```

This will result in ATS running andyroid and then, if successful, andyroidPoster, with two different labels and values for delta.

We used the value of the argument delta to set the command line arguments, but we also set it as a test option. Then, if we want to run only those tests with `--delta`, we can do it with a filter:

```
ats -f 'delta' suite
```

We might also choose to modify this example to include `group()` and `endgroup()` at the top and bottom of runAndPost; the group call could set options that we wanted in each test statement, and we would save all the output in case of failure of any part of it.

Leveling

ATS has a built-in leveling filter. Using `stick` to set a value, you can break up tests into levels and execute only those below a certain value, or between certain values:

```
source('andyroid/andyroid.ats')
andyroid = get('andyroidTest')

stick(level=10)
andyroid('test1.in', label='test1')
andyroid('test1.in', label='test1d', delta=True)

stick(level=20)
andyroid('test2.in', label='test2')
andyroid('test2.in', label='big run', delta=True,
        level = 30)
```

executed with command-lines such as:

```
ats --level 10 suite
ats -f 'level >= 4 and level =< 12' suite
```

These two levels, 10 and 20, might correspond to daily and weekly tests for example. We recommend leaving some room to change your mind.

Introspection

When a file is sourced, the normal action is to execute the contents of that file using the ATS vocabulary. However, magic is possible! Before explaining how to do the magic, let's understand the motivation for it.

An expert on the some area of *andyroid* may have a test routine, say "testX.in", that he uses, running the program with some variety of test inputs. However, another member of the team will in general not know how to do this. So it would be nice if the expert had a good way to embed in the test the knowledge of what parameters to use to run it, without interfering in the ability of the expert to run it by hand in a different way.

Now, strictly speaking you don't have a problem here. You can have a separate test file, testX.ats, and this file can have test lines for each test the expert believes should be run, perhaps also giving them appropriate levels, time limits, etc.

However, that often leads to having this extra file for absolutely no reason other than to make this information available to ATS. And so is born the concept of "introspection": ATS looks inside a file it is about to source and discovers that it is both the input file to be tested and the instructions on how to test it, the latter appearing to be comments.

For example, assuming *andyroid* uses a "#ATS:" at the start of the line to denote magic comments, testX.in might look like this:

```
#ATS:andyroid(inputFile=SELF, label="testX easy")
#ATS:andyroid(inputFile=SELF, delta=True, label="testX hard")
... body of the testX file
```

This would cause source("testX.in") to actually create two tests, where the word SELF will evaluate to "testX.in". The file will not be further sourced, so the language used in the rest of it need not be Python.

If you wish to source a file with a different magic commenting convention, this is possible – see the User's Manual explanation of the `source` function.

Putting It All Together

The example given here in file `suite` is expanded in file `fancySuite`. There you can see use of many of the concepts discussed here and in the advanced section below. Note that the file begins by sourcing that same andyroid.ats file that we used before. It then starts the testing with Test/main.ats, which in turn sources files in directories simple, delta, and psweep.

In directory `simple` there is a test that is going to fail. It has been given an option "development=True". A default value of False has been given to the other tests by using a `glue` statement in main.ats. Since we used `glue` and not `stick`, this value persists into the subdirectories.

In directory delta there are some tests that turn on the `--delta` option.

In directory psweep we see that in fact psweep.ats is the input file for Andyroid, but introspection is used to execute it many times with different values of alpha.

Advanced Topics

Modifying ATS itself should rarely be necessary. The techniques in this chapter show how much you can do with customized drivers and machine specifications.

Expecting Failure

Applying the tilde (~) operator to a test marks it as a test that is expected to FAIL. Thus:

```
~test(....)
```

will be considered to have passed only if it ends up with status FAILED. The status will be changed to EXPECTED and a entry made in the t.notes list documenting this fact. See the reference manual for further details.

Postprocessing

Postprocessing the results of the ATS run can be done using a custom driver or using the `onExit` facility.

Using the Log

One of the defined vocabulary items is an object named `log`; it acts like a function and prints its arguments into the log and / or on the terminal, space separated and terminated by a newline. For example, if you put this in your sourced file:

```
log('Entering test section', 'foo', echo=True)
```

then "Entering test section foo" will be printed to the log and to the terminal. This may give you a good fuzzy feeling if you are unsure of what tests are being initiated.

The `echo` value controls output to the terminal. Another flag, `logging`, controls whether or not the output is stored in the log.

Using Magic Output

When a test writes something to its standard output that begins with some magic prefix, ATS captures those lines and stores them in the test object as a list (`test.output`). The lines have their final newline removed. If the option `hideOutput` is True, such output is written in the log when the test finishes. By default it is False.

The magic output prefix is set in the test's option `magic`; the default value is `#ATS:`.

Note: This differs from the `magic` argument to `source`; they have the same default but otherwise are not connected. The `source` magic controls the introspection process for input; the `test` magic option controls the capture of part or all of the output from running the test.

Setting the `magic` output prefix to `None` prevents any output collection.

The list would be available to any post-processor using `onExit` or a custom driver. You may wish to set `--hideOutput` if you are just going to post-process.

Output Magic Example

If we define a test with a magic option of "shazam!":

```
test1 = test(executable=something, ..., magic="shazam!")
```

Suppose the test runs and prints:

```
shazam!4.2 6.8
```

After the test exits, `test1.output` is `["4.2 6.8"]`.

Capturing All The Output

Any test with a `magic` option which is an empty string (formed by using two consecutive single or double quotes) then all the output from the program is captured and stored in the test's `output` attribute. You can then do something with it via postprocessing or view it in the log.

A Note on Notes

Each test also has an attribute `notes`, a list of strings. These notes are currently used to note that certain things have happened and are used in the summary of results. You can append strings to this attribute if you wish.

Making Custom Drivers

The main program `ats` is a very short script; stripped of some error reporting it reads:

```
#!/env/bin/python   [this line adjusted on installation]
import ats
ats.manager.main()
```

`ats.manager` is an object that controls the ats run. Before you call `main`, you can do other things such as register `onExit` functions. You can massage the arguments (`sys.argv[1:]`) and pass the resulting **string** as `main's` argument. (Python's **shlex** module can help manipulate argument lists.).

After `main` returns, the master list of tests is `ats.manager.testlist`. All the statuses are available as attributes in the ats module.

For example, this driver would add a list of tests that failed in some way to a database:

```
#!/env/bin/python   [make sure it points to ATS's python]
import ats
from ats import CREATED, INVALID, FAILED, TIMEDOUT, manager
manager.main()
failed = [CREATED, INVALID, FAILED, TIMEDOUT]
(open the database)
for test in manager.testlist:
    if test.status in failed:
        (write test.name and details to database)
(close the database)
```

The task of installing this script alongside the main ats script, and adjusting the first line, can be handled with a separate setup.py script or by editing setup.py to add another script before installing. Change this line in setup.py:

```
scripts = [codename]
```

to read:

```
scripts = [codename, "your_script_name"]
```

If you need tighter control, instead of calling main you can call its constituent parts:

```
from ats import manager
manager.init(clas)  # note, string argument
                    # omit clas = use command line
manager.firstBanner()
manager.core()
manager.postprocess()
manager.finalReport()
manager.saveResults()
self.finalBanner()
```

Here is what those pieces do:

- `init` processes the command line and the machine gets defined. The function has 3 possible arguments: a command line, and two call-back functions for adding and examining command-line options.
- `firstBanner` initializes the log – before this has been called, using the `ats.log` object will just write to the terminal. After this call, the manager vocabulary is "up", so you can safely call things like `glue`, `define`, `test`, etc.
- `core` does the "phases" of collection, sorting, and execution.
- `postprocess` calls the user's onExit routines.
- `finalReport` writes the detailed report.
- `saveResults` creates the *atsr.py* ` file.
- `finalBanner` writes ATS' summaries and exit messages.

Please let the authors know of any needs for further refinement.

The handyAndy Custom Driver

Andyroid has a little custom driver `handyAndy`. This driver takes care of the sourcing of `andyroid.ats` and does some postprocessing looking for failures that did not have the the option `development` set to True; these are true failures. If users used `handyAndy` instead of ats itself, the `source` - `get` procedure could be left out of all the ATS input files.

1.4.2 Reference Material

Test Selection and Execution

Tests are defined in ATS input using two commands, `test` and its little brother, `testif`. However, not every test that gets defined is necessarily going to be executed. The user can define logical conditions (*filters*) that a test must satisfy to be chosen for execution, and the hardware available may cause others to be skipped.

In order to make it easier to structure suites of tests, there is an elaborate set of facilities involving filters, command-line options, and arguments to `test` statements, as well as facilities for grouping and ordering your test executions.

ATS Execution and Command-line Options

In specifying the names of input files, you can give the filename or omit the filename extension. ATS will attempt to find the file using its name, then with a `.ats` extension, and then with a `.py` extension.

Unix or Mac

To start ATS on a Unix system or Mac, execute this line in a terminal window:

```
ats [options] [input files]
```

Note that the `--exec` option is frequently used to define a default executable, but any given test can specify any executable as the program to be tested.

Before executing ATS, it may be desirable to have defined the environment variables MACHINE_TYPE and / or SYS_TYPE; and there may be others for testing particular executables. Please consult with the owner of your local ATS installation, and the owners of any custom ATS drivers you may be using.

Windows

Execution on windows can be done in the same way from a command window, but can be made more convenient by defining a `.bat` file, such as:

```
C:\python27\python c:\python27\ats $*
```

These instructions need improvement as the first Windows users determine the right way to do this.

Command-line Options

What follows are the the most important command-line options available in any ATS installation.

Note: The exact set of command-line options depends on the machine you are using and / or upon any custom driver you are using for testing a particular program. To see the complete list for a given ATS installation, enter `ats --help`.

This will also show you abbreviations for some of the options.

--allInteractive Run every test in interactive mode.

--cutoff cutofftime Over-rides the timelimit for all jobs no matter where the timelimit is set. Jobs that fail once reaching the cutoff will TIMEOUT. The forms for giving the time are the same as for `--timelimit`. Note: Jobs that TIMEOUT are marked as FAIL when using Flux.

--debug Debug mode; prints more information in the log and on the shell window.

--exec EXEC Give the path to the code to be tested. The path is tilde- and dollar-expanded.

This option sets the environment variable ATSROOT, if not already set, to the directory in which the executable resides. Most of the time this option is used, and the executable so named is referred to in this documentation as the *specified executable*.

However, tests with different executables can also be specified, by using the executable='/path/to/my/code' as one of the test options. The purpose of ATSROOT is to allow you to specify related tools for your code that are located in the same directory as the executable. In specifying a test, you can use this variable in the script or executable using either *$ATSROOT* or *%(ATSROOT)*.

Note that you don't have to have one main code to be tested. You can specify a different executable for each test, or group of tests.

--filter FILTER Add a filter; may be repeated. Be sure to use quotes if the filter contains spaces and remember that the shell will remove one level of quotes.

--glue FILTER Has the effect of executing *glue(FILTER)* before execution of the tests. May be repeated. Be sure to use quotes if the filter contains spaces and remember that the shell will remove one level of quotes. The glue function is used to set persistent test option defaults.

--help Show the list of options and exit. There may be more options than are shown in this document, such as batch or node control options.

--info Print information about ATS, such as version, path to the executable, and some parameter values.

--keep Keep the output files from the tests that succeed. Normally the output from tests that fail, or which must be checked, is kept.

--logs LOGDIR Sets the name of the log directory. The default log directory is *arch.time.logs*, where arch will be an architecture-dependent name, and time will be digits of the form *yymmddhhmmss*. All logs and the continuation file are placed in this directory. The log itself is named *ats.log*.

--level LEVEL Set the maximum level of test to run. Level is simply a built-in easy-to-use filter.

--skip Skip actual execution of the tests, but show filtering results and missing test files, and show additional details about the input.

--nobatch Do not run any "batch" tests..

--nosrun Run the code without srun. This option can also be used on BlueOS to run ALL test on a login node as it circumvents the login node check. If the tests need to be run on a working node, then the tests themselves will need to get an allocation.

--npMax value Value is an integer, the maximum number of tests to run at once (on a node, if multinode machine). Some machines allow you to set this higher than the actual number of nodes, at your own risk.

--okInvalid Run tests even if there is an invalid test. Examples are tests specifying missing scripts or executables.

--oneFailure Stop if a test fails.

--removeStartNote Removes the messages printed at the start of a test running.

--removeEndNote Removes the message printed at the end of a test running. Will still get results printed with pass/fail, just no "stop".

--serial Run only one job at a time.

--timelimit TIMELIMIT Set the default `timelimit` test option. TIMELIMIT may be given as an integer number of seconds or a string specification such as '2m', or '3h30m20s'. A similar notation can be used for filtering by time limit, such as *-f 'timelimit < "30m"'*. Note: Jobs that TIMEOUT are marked as FAIL when using Flux.

--verbose Verbose mode. Both starts and finishes of tests are noted on the terminal, plus other reports. Test failures are reported regardless.

--version Show program's version number and exit.

Basic Operations

The goal of ATS is to execute a series of test problems. It does this by reading input files written in the Python language, with some predefined ATS functions added. In particular, ATS supplies a function named `test`. Each execution of the `test` statement defines a particular program to execute, including its command line and a variety of options used by ATS to know how to run it or to decide not to run it.

After running the tests, the ats prints a summary of which tests have passed (that is, returned with a normal exit status) and which have failed.

The second basic statement is the `source` statement, which causes a file to be read containing additional commands. An introspection procedure, described below, is also available to allow scripts meant as problem input to contain definitions of how they are to be run when run by ATS.

Retrying Failed Tests

If any tests fail or are not completed, a "continuation" file is written and a message issued in the summary section giving the name of the file. The continuation file is named continue.ats and it is inside the log directory.

You can rerun the exact same ATS command, adding the path to the continuation file as an extra command-line argument.

Note: You must run the *exact* same command with this added argument at the end of the command line.

Doing this will redo those families of tests that had a failed member. This process may be repeated until all tests pass. In your log, tests that had passed before well be marked "Previously passed" and batch jobs will be "skipped". The continuation file is pretty self-explanatory and you can edit it with thought.

Note that if a descendent of a test failed, the test will be rerun because the error might have been in files produced by the parent test, even though it appeared to pass.

The intention of this facility is to let you fix your code without having to rerun all your tests. For correctness, you should rerun everything once you believe you have corrected all errors.

Results Facility

Each run creates an `atsr.py` file in the log directory. This file, if run under Python, creates one variable named "state", which is an object that is a dictionary whose values can be read and written using either dictionary or attribute notation. This type is called an AttributeDict.

The object state has attributes corresponding to the major features of the manager object, including a `machine` and `testlist`, which is a list of AttributeDicts, each encapsulating the major properties of each test.

Two methods in the manager object control this facility, which may be used by custom drivers.

`onSave`(*saver*)

> Registers a `function saver(results, manager)`, which will be called when the data for the state is collected. It may modify the AttributeDict `results` in any way it likes, usually by adding to it. Calling `results.clear()` would be a way of minimizing the use of resources devoted to this file.
>
> onSave is available in the test environment also, for use in input files. Note that the call does not cause the save of the file at the time it is executed.

Three other manager methods can be called from custom drivers.

`getResults`()

> Returns the AttributeDict containing the state. The manager's `machine` and, if set, `batchmachine`, are given a change to contribute fields to the end result, and finally any onSave-registered routines are called in the order they were registered.

`saveResults`(*filename='atsr.py'*)

> Save the state to a file using given file name; if not absolute, put it in the log directory.

`printResults`(*file=sys.stdout*)

> Do the actual job of writing the state file. Here file should be an open file handle. You would only use this function if you wanted to add something to the file other than the `state` variable.

Normally `saveResults` creates the file and asks `printResults` to call `getResults` and print the returned state into the file, preceded by a header that imports the symbols in the `ats` module so that the code will execute correctly.

Interactive inspection of the resulting file is most easily accomplished with an interactive Python session, such as:

```
cd <logdirectory>
python -i atsr.py
   print "Number of tests = ", len(state.testlist)
   print "Machine name", state.machine.name
   print "Number timed out", \
         len([t for t in state.testlist if t.status == TIMEDOUT])
```

Note that ATS statuses will compare equal if they compare to another status or the name or the abbreviation. So in the last line above, TIMEDOUT, "TIME", or "TIMEDOUT" would all work.

To compare different files you can rename state as you read it:

```
d= {}
execfile("atsr.py", d)
state1 = d['state']
```

You can change the name of the file to be used by setting manager.saveResultsName in your input file. If not an absolute path, the file will be created in the logs directory.

Controlling Input

File Sourcing

source(**paths*, ***vocabulary*)

Process one or more paths as if each was the name of an input file given on the command line. (This function is the same as manager.source)

The current stuck options are saved upon entry, cleared before beginning processing, and then restored on completion. See *stick* below for further details.

Path names are expanded both for tilde and environment-variable names using the dollar sign.

The vocabulary items can be any number of keyword = value pairs.

Vocabulary words are added to the environment in which input files are compiled by Python. The scope of this environment is just within the input of the paths given to this source command. To add a vocabulary value to all subsequent source commands, use the *define* command, described next.

The vocabulary word *introspection* can be used to change the commenting convention used for ATS' introspection facility. Details are given below.

define(*keyword*=*value*, ...)

adds one or more keywords to the vocabulary used by the source command to parse input. This is the same function as manager.define.

undefine(*keyword*, ...)

removes one or more keywords from the vocabulary used by the source command to parse input. This is the same function as manager.undefine.

showDefine(**keywords*, ***options*)

logs the current definition of one or more keywords in the vocabulary used by the source command. If no argument is given, all the definitions are shown. This function is used to help debug your vocabulary setup. The options may include echo and logging, and are passed on to the call to log. The defaults are both True. This is the same function as manager.showDefine.

A file may be 'sourced' because it was given on the command line or because a `source` function was executed with it as an argument. (Note: In what follows it is is assumed that a line that starts *#ATS:* is a comment to your application; however, it is possible to change the commenting convention to suit your input convention, using the second argument to `source`.

Examining and prioritizing tests

After collection of the tests the user may wish to examine or alter the tests before they are executed. This is done by registering one or more routines to be called (in the order in which they were registered) by using `onCollected`. See also `onPrioritized`, below.

`onCollected`(*routine*)

The routine registered is called when the input is complete. It is given the manager object as its single argument. The routine thus has access to the `manager.testlist`.

The routine may make use of the routine that ATS itself is about to use to divide the tests into interactive and batch tests:

```
interactiveTests, batchTests = manager.sortTests()
```

You can effect what happens next by changing statuses (such as setting the status to BATCH or FILTERED or CREATED (i.e., interactive)) or change `totalPriority` (see below).

You also have a chance at this point to use each test's `directory` attribute to prepare the file system, or to build data structures for later use in a postprocessor.

Use this facility with caution. Do not attempt to change tests that would not have executed at all into ones that will. If you change a label it must be unique when you are done. Do not alter serial or group numbers.

After the `onCollected` actions, the scheduler prioritizes the interactive tests. The `totalPriority` attribute of each test is set to the sum of the test's own value plus the sum of the priorities of each test that must wait for this one to complete. (Such conditions are created by dependencies or `wait` or `group` commands.)

The user may wish to examine or alter the priorities of the tests tests before they are executed. This is done by registering one or more routines to be called (in the order in which they were registered) by using `onPrioritized`.

`onPrioritized`(*routine*)

The routine should take a single argument, interactiveTests. The intent is for the user to examine or alter the `totalPriority` attribute of a test. Altering `priority` attributes will not work. Altering anything else about the test is probably ill-advised.

In summary, there are two ways to change the `totalPriority` attribute: in an `onCollected` routine, which will contribute the new value to its predecessors, or in an `onPrioritized` routine, where you are setting the final absolute value.

Using Introspection

When a file is sourced, ATS looks to see if the file contains any lines that begin with the five characters *#ATS:*. If so, the set of such lines with the leading *#ATS:* removed will be executed as Python code. The remainder of the file will be ignored. This procedure is called *introspection*.

Note that Python's indentation rules apply, so there should not be any spaces after the *#ATS:* except on lines that should be indented.

For example, continuation of lines is allowed in the normal Python manner:

```
#ATS:test('myfile.py',
#ATS:     'my command line args',
#ATS:     np = 4)
```

Picture the first five characters as defining the left edge of the lines to be executed.

During this procedure, the symbol SELF will be defined to be the name of the file being sourced. Thus a line such as:

```
#ATS:test(SELF, 'command line options', np=4, w=2)
```

will cause the file to be tested with the given command line, using the options np = 4 and w = 2 as context for filtering.

A file may contain many such lines, in order to exercise the same test with a variety of parameters. Also note that not all the *#ATS:* lines need to be ATS commands; they can be any Python code. They can also include log commands, source other files, etc.

Changing the introspection convention

If a value for the vocabulary word "introspection" is given, it should be a python function which, when given a line, returns None or the value of the line as introspection. The default is a function that returns None unless the line begins with *#ATS:*, in which case it returns the line less that prefix.

By prescribing your own value for introspection, you can allow the introspection process to work on source files with a different commenting convention than "#".

In particular, to change the default function used for introspection, just use define after you declare it. For example:

```
def asteriskinterpolation(line):
    "Any line that starts with *ATS: is magic"
    if line.startswith("*ATS:"):
        return line[5:]
    else:
        return None
define(interpolation=asteriskinterpolation)
```

Grouping

If you have a test that creates some files for postprocessing, you can group that test with the related ones.

You begin with:

group(*independent=False*, *report=False*, ***kw*)

and after defining some tests, finish with:

endgroup()

A group is also ended by another group statement, or the end of the current input file. The arguments to the `group` call become default options for each test defined inside the group. They can be overridden by options in the `test` and `testif` statements within the group.

Only the first test result will be included in the final reports unless some member of the group fails, or you change the report argument to True. The output files of the entire group will be kept if anything fails; otherwise the usual keep options will prevail.

The `independent` test option determines if a test will block any other test (other than ones in its group) that uses the same directory. By default, then, a group will lock-out any non-independent test or group from running in the directory or directories its tests use. This is not different than the default behavior of ATS, but is a convenience for making sure that the members of the group will not be interleaved with other, non-independent tests that use the same directories, if you have glued or tacked or stuck independent to be be True.

These two arguments are used as test options for all tests in the group, but for any particular test can be overridden by an explicit option in the test statement itself.

Note that grouping does not make each test depend on the preceding tests in the group. Two members of the group may execute together. It also does not make the failure of one test skip another. To achieve dependency, use the 'testif' facility.

Wait

It is certainly possible to make two tests that appear to be independent but which cannot in fact run simultaneously. ATS prevents many cases of this due to its reluctance to run two tests in the same directory at the same time. If that fails to solve the problem, and the `group` or the `testif` statements are not sufficient, you can try the `wait` statement:

`wait`()

> All the tests defined so far in this source file will be finished before proceeding to any tests defined later in this source file. Tests defined in other files that are sourced *after* the 'wait' must also wait for all the tests before the wait in this source file.

wait() may be a useful way to express massive dependencies without using excessive *testif* calls. However, if used excessively, *wait* may cripple ATS's ability to run tests simultaneously.

You can debug your wait structure with this command:

```
ats yoursource --skip
```

This will show a list at the end of the log file, under "ATS RESULTS", showing the serial numbers being waited for by each test.

When all tests are completed, ATS issues a final report and runs any postprocessors that have been registered using the *onExit* facility described later.

Example

Suppose we have this test file "waitforit.ats":

```
glue(executable = "/bin/ls")
test(label='first')
test(label='second')
wait()
test(label='third')
```

Then the third test will not execute until the first two are done – but this says nothing about the order in which the first two will execute.

Suppose now we add a source of another file, so we have:

```
glue(executable = "/bin/ls")
test(label='first')         #1
test(label='second')        #2
```

(continues on next page)

(continued from previous page)

```
wait()
source('waitfor1.ats')
test(label='third')         #6
```

with the file being sourced containing:

```
test(label='waitfor1 first')   #3
test(label='waitfor1 second')  #4
wait()
test(label='waitfor1 third')   #5
```

We have thus defined six tests in all. The output of the debugging process is:

```
Interactive tests:
#1 INIT ls(first) ready
   []
#2 INIT ls(second) ready
   []
#3 INIT ls(waitfor1 first) ready
   [1, 2]
#4 INIT ls(waitfor1 second) ready
   [1, 2]
#5 INIT ls(waitfor1 third) ready
   [1, 2, 3, 4]
#6 INIT ls(third) ready
   [1, 2]
```

The parts in square brackets are lists of the tests this one must wait for. (The list will include any tests of which this one is a dependent.) So we see for example that `#6`, the last test in the main file, waits for the first two tests, because a `wait()` occurs after `#2`, but it is not affected by the wait statement in the sourced file. In that file the first two tests are waiting for the first two, and the third waits for the first four.

Executing Tests

ATS attempts to execute as many tests as it can at the same time in order to keep the computational resources it has been given busy, subject to respecting the test options `priority` and `independent`, and the `group` and `wait` statements. The following sections describe this process.

Scheduling

After the ATS has read all the input and knows what tests are to be run, it examines the collection and combines the information generated by the *group*, and *wait* commands with the test dependencies to figure out which tests must execute before others. It can then combine the priorities of tests to determine a preferred order of execution – which however will be subject to processor availability.

This work is done by a scheduler object. A standard scheduler is provided, and is an attribute on the `machine` object. A user could potentially modify it by inheritance from its defining class, `schedulers.StandardScheduler`.

Each test has a priority. By default the scheduling priority (`totalPriority`) is the number of processors required by the test plus the priorities of any tests which cannot execute until this one is finished. In this way those tests with a lot of dependents are started early.

A test may specifiy its priority as an option "priority=n" where n is a nonzero integer. A test whose priority is zero or less will not be run. Thus, a long-running 1-processor job without dependents might profit from being given a priority, say 3, so that it starts earlier. Note that an np = 0 job requires 1 processor.

As tests are selected to be started, the highest-priority job that will fit on an available machine is chosen. You can examine the tests in postprocessing if you want to understand what influenced the scheduling:

- Test option priority,
- Test attribute totalPriority,
- Test attribute group,
- Test option independent (described below)
- Test attribute runOrder, an integer indicating the order of test launch.

Note: Important: by default two tests will not be run in the same directory at the same time.

This is a modestly conservative scheme to avoid common resource conflicts when testing one file with different parameters.

If you know a test does not have such a problem, you can give it the option `independent = True`. Note that the `group` command makes the default value of `independent` False for all members of the group, overriding anything except an actual option in the test statement. Thus if you do not want this behavior for the group you must use independent = True as an argument in your group command.

The standard scheduler sorts the groups by the highest priority test in the group. In effect, every member of a group behaves as if it has the priority of the highest-priority test in the group. This ensures a large prejudice towards running members of a group once it has started, until they are all complete.

Progress Reports

When a test starts this fact is shown on the terminal output. You can use the command option `--verbose` to cause test completions and other additional events to be reported as well. All the information is always in the log. Additional output is generated by the `--debug` option.

Every minute ATS issues a report on its progress to the terminal only.

Output Files

The standard output and standard error of a test are written into files in the directory where the logs are written. These files are (usually) removed when the test concludes successfully; for a group, this occurs when *all* members of the group have succeeded.

The name and label of the test script or executable, along with the test's serial number, are used to create the file names.

The –keep option prevents the removal of these output files even when the tests are successful. They are also kept if the test has the option keep=True or check=True.

See also:

Postprocessors set using the *onExit* facility can access the magic output of a test as test.outputats.

Interrupting a Run

A control-C interrupt will terminate the program and all the tests it is running. Any test started but still not finished will be reported in RUNNING status.

Creating and Selecting Tests

Creating Tests

test(**args*, ***options*)

This notation means that you can give positional, unnamed arguments, followed by keyword=value arguments.

- If you give just one positional argument, it is called "script".
- If you give two, they are "script" and "clas".
- If you do not give one or both positionally, they are given in the options, with their default values being blank strings.

It is an error to give more than two positional arguments.

Positional arguments are allowed for backwards compatibility – it is preferable to name everything.

In the test function call:

- script is a file name, which may be be relative to the directory containing the input file or absolute. Note that ATSROOT can be used in such names to designate either a preset environment value or the directory of the specified executable. The script if given will be used as the first argument on the test's command line, and will supply a default name for the test.
- clas is a string giving the command-line arguments to be passed to the execution. Before doing so, python string interpolation is used with the options dictionary. This means, for example, that:

```
test(clas = "-in %(input)s -parallelism %d", np=4, input='foo')
```

will result in:

```
clas = "-in foo -parallelism 4"
```

You might want to do this if, for example, this expression for clas was constant over many tests except for these variations of input and np. Then you could stick or glue this value for clas and not have to repeat it over and over.

Options can be any keyword = value pairs declaring the properties of this particular test; these are used in filtering and also serve as documentation for the test's properties.

test returns an test object whose attribute 'status' is one of the following attributes of the ats module: CREATED, RUNNING, HALTED, PASSED, FAILED, TIMED, FILTERED, SKIPPED, BATCHED, INVALID.

Warning: Testing the truth value of a test object, such as using it in an *if* clause, causes the test to be marked FAILED. See *testif* below.

The test object will execute in the directory *test.directory*. This value can be set in the test options, but if it is not (which is usually the case) it is set to the directory in which the script resides, if the script is given. Otherwise it is set to the directory in which the test statement was read.

Note that if executable is 1, the script isn't really a script, so directory is set to the directory in which the test statement was read.

testif(*othertest*, **args*, ***options*)

This is the same as the test statement except that this test will only be run if `othertest` is eligible to run, has been run, and has been successful.

For example:

```
t = test('foo.py', 'dumpat=25')
testif(t, 'foo.py', 'restartat=25', label='restart test')
```

Explanation: This works because the test call returned a test object, `t`.

Expecting Failure

Sometimes you want to make sure a test will fail. To do this use the tilde (~) operator on the test:

```
~test(....)
```

The test will count as passed if its status ends up FAILED.

You can also set the `expectedResult` attribute of the test directly to something other than PASSED:

```
t = test(....)
t.expectedResult = TIMEDOUT
```

It is pointless to have a dependent of a test that is not expected to PASS. It will be SKIPPED.

Test Options

Each test can define arbitrary keyword = value pairs. With the exception of a few special options described below, the keyword names are arbitrary. Most options do not affect the running of the test, just the decision about whether or not to run it.

There are five lifetimes of option specification:

- defaults (often with command-line options to change the value),
- permanent (see glue and unglue),
- current and descendent files (see tack and untack)
- per sourced file (see stick and unstick), and
- per test (using the options portion of the test command).

Reserved option names

While you are free to use any desired scheme for options and filters, do not use the following names except for the purposes described. These are listed roughly in the order of their frequency of use by the end user.

label

label can be set to a string that will be appended to the name of the test to identify the test more fully. Thus, two different runs of the same script can be distinguished. label by default is the test's serial number, the number that distinguishes the order in which the test was defined. labels are adjusted after all tests have been read to add distinguishing characters, so that no two tests have the same label.

name
This is the test name, as is printed out in the summary. If a script is given, it is that file name less the extension. Otherwise it defaults to the base name of the executable.

np
The option 'np' is reserved for specifying the number of processors to be used to run the program if the machine is a parallel processor. np = 0, the default, means a scalar run. np = 1 will be treated as a serial run on serial computers. np can be used in filters, e.g. *np < 32*.

executable
This option sets the path to the program to be run for this test. The default value of this option is usually set by the –exec command line option.

The executable program will be considered to have passed or failed depending on its exit status.

The executable may contain options after the path; it may also be given as a list of strings, the first component being the path and the rest options. If the path contains an internal space, you must use the list form.

Deprecated since version If: executable is 1, the first positional argument to the test function is the name of the executable program. It is preferable to use *executable = /path/to/executable*.

batch
This option is used to run a test in batch by setting it equal to 1 or True. Note that the filter *batch* (which you can set with the –filter batch command-line option) will restrict submissions to only batch jobs and the remaining non-batch jobs are skipped.

check
If check is not zero, this test is marked to be checked by hand rather than marked as passed, if it finishes normally. Such jobs are reported separately in the summary.

keep
If true, the test's output files are kept even if it passed.

independent
If independent is True, the user is certifying that there is no obstacle to this test executing at the same time as any other test. Otherwise, by default tests are assumed to conflict with others in the same directory, because they might write files there with the same names as those read or written by other tests. If two tests conflict, they are never run at the same time. Judicious use of independent = True will increase ATS's throughput. We suggest that while a stick(independent=True) may be appropriate, in some test files, to glue this definition may be reckless.

priority
By default the priority of a test is np + the sum of the priorities of and dependent jobs. The priority option lets you override this by giving an integer value. A value of zero means the test will be skipped.

env
By default the environment passed to the test will be the value of the ATS environment `os.environ`. To modify this dictionary, give the option env=D, with a value D that is a dictionary of the additions or changes to environment variables that you desire. If None, or not given, the default is used.

record
If a test is given option record=False, it is not reported as a separate test unless it fails in some way.

timelimit
Specifying a timelimit denotes maximum execution time for the test. For example, timelimit="30m" will kill the test after 30 minutes and give it TIMEDOUT status.

SYSTEMS
SYSTEMS defaults to a list of one value. That value is the value of the "name" attribute of the machine object ATS has discovered. A filter:

```
  s in SYSTEMS

where s is this same value, is always used. Thus, by specifying SYSTEMS as
an option, the test will run only on the machines(s) named in SYSTEMS.
```

magic
: magic controls the treatment of certain lines of test output. The default value is `#ATS:`.

  If a test prints any lines beginning with the characters *#ATS:*, those lines will appear verbatim in the output, but also will be printed, less the *#ATS:* prefix, in the summary messages that appear when the test finishes.

  If magic is set to None or a blank string, the entire parsing of the output file is skipped.

hideOutput
: If true, do not print magic output lines in the log.

same_node
: ONLY WORKS ON FLUX (ATS can run Flux under slurm). Specify a string identifier for the tests that you want to be run on the same node. Useful for tests that depend on some data output by another test that might not be accessable from other nodes. NOTE: Using this option will limit -N 1 and -n to max on one node, if more than that were requested. Ex: same_node='abc'

Extra Arguments On The Executable

If you want to always execute a given application with some fixed arguments in addition to others that vary, you may give them as part of the executable option to a test or on the command line. For example:

```
my_application = "/foo/bar -a -b"
test(clas="-d", executable=my_application)
```

will result in the execute line `/foo/bar -a -b -d`.

Be careful about quoting levels. For example, to make a test that did the equivalent of:

```
python -c "print '3+4'"
```

you must use an extra quotation level:

```
my_application = "python -c"
test(executable=my_application, clas = "\"print '3+4'\"")
```

Filters

A filter is a string that can be evaluated to a logical result. Filters can be defined with the command line option -f or –filter, or using the function *filter*. Helper functions can be defined using *filterdefs*.

Each test declares options: these are keyword = value pairs. To decide whether or not to execute a test, each filter is evaluated using Python's eval function, in an environment consisting of these symbols:

- The options set by the test (including current 'stuck', 'tacked', and 'glued' option values described below)
- Symbols created parsing of text added by calls to filterdefs.
- The ats environment, consisting of these objects, which are each described in this document:

```
manager, test, testif, source, log, filter, filterdefs, stick, unstick,
tack, untack, glue, unglue,
getGlue, getTack, getStick, sys, os, AtsError, AtsTest, abspath,
is_valid_file, is_valid_executable, statuses,
CREATED, RUNNING, INVALID, PASSED, HALTED,
FAILED, BATCHED, SKIPPED, FILTERED,
SYS_TYPE, MACHINE_TYPE, MACHINE_DIR, BATCH_TYPE,
onExit, onSave, getResults.
```

- SELF is equal to the test object and some of its attributes may be interesting for filtering (name, label, basename).

If the filter returns true when evaluated, the test will be run. Otherwise, or if the filter gets a NameError when evaluated, the test will not be run.

Thus, a test run with:

```
test('mytest.py', x = 7)
```

would pass the filter 'x==7' but not pass the filter 'x==5' nor the filter 'y==7' (because the symbol y is not defined by the test).

Additional ATS Vocabulary

ATS input is written in a expanded dialect of Python. That dialect contains the following facilities.

Debugging and logging

debug([*value* = *None*])

debug() can be called in your input; it will return the current debug level: zero if –debug was not specified, or one if it was.

You can give debug an argument to set a new value, such as debug(2), and issue conditional code depending on the value which is returned by debug().

log(**items*[, *echo=False*, *logging* = *True*])

The log written by ATS, and the terminal (in the form of stderr), can also be written to from user input. The log function adds a line to the log, using the enumerated items as if in print statement, unless logging is False. If echo is True, it prints to standard error.

With no items log prints a blank line.

For example:

```
log("I want to eat", 5, "donuts")
```

prints:

```
I want to eat 5 donuts
```

terminal(**items*)

This is a version of `log` that writes only to the terminal.

Other methods and attributes in the log object are:

`log.indent()`

Increase the current indentation.

`log.dedent()`

Decrease the current indentation.

`log.reset()`

Reset indentation.

logging

A switch that controls logging to file

echo

A switch that controls logging to stderr.

Shortly after it gets organized, log sets the defaults for logging and echo. To be SURE you write something to stderr, use echo=True. And if you change logging or echo, or the indentation level, put things back as you found them, please.

It is not possible to log a partial line.

Manipulating Test Options

The following facilities provide for setting more-or-less persistent default values for test options. Each type listed will override the ones above it while it is still in scope.

1. A default value for most options is built in to ATS.
2. Command-line options override the default. Command-line options are not available for every test option, just the most important ones.
3. glued: Values set with a *glue* call. Such values apply until overridden by another glue call.
4. tacked: Values set with a *tack* call. These values apply until processing of the current file is finished, including in files sourced by this one.
5. stuck: Values set with a *stick* call. These values apply only in the file in which the call appears.
6. group: Values set with a *group* call. Such values can be overridden by an explicit value in the test. Group values last until the next `group` or `endgroup`, or the end of the source file.
7. explicit: Options given in a *test* or *testif* call always apply to that test.

Great care should be used with glued and tacked options, because they are not visible locally in files that are later sourced "from above", and a person working on one of these files may not realize they are inheriting a value already that will take effect unless they override it. This will also cause the file to behave differently if used stand-alone as opposed to sourced from another file. Use the least scope that will get the job done for you.

Putting tests in groups has other consequences you should be aware of. See in particular *directory blocking*.

Here are the functions for controlling test option defaults:

stick(***keys*)

Add the keyword = value pairs to the current dictionary of stuck test options. Stuck options persist until the end of the current file but do not apply in files sourced from this one.

A stuck option overrides a tacked or glued option, and is in turn overridden by an explicit option to `test` or `testif`.

tack(***keys*)

Add the keyword = value pairs to the current dictionary of tacked test options. Tacked options persist until end of the current file and do apply in files sourced from this one.

A tacked option overrides a glued option, and is in turn overridden by a stuck value or by an explicit option to `test` or `testif`.

glue(***keys*)

Add the keyword = value pairs to the current dictionary of glued test options.

Glued options apply to all subsequent test definitions. A glued option can be overridden by a stuck or tacked option, which in turn can be overridden by a value given in a test or testif statement.

> Think of glued options as permanent changes to the default value of an option. One use might be to be sure every test has a value for some option name so that a filter can be constructed.

Notice the language here carefully. In the following example, the value which will be used in the test for the option color is "blue":

```
stick(color = "blue")
glue(color = "red")
test("myscript", clas = "%(color)s")
```

The stuck option overrides the glued one of the same name.

Items can be removed from these dictionaries with:

unstick(**names*)

Remove each name from the list of stuck options. If no list is given, remove all the stuck options.

untack(**names*)

Remove each name from the list of tacked options. If no list is given, remove all the tacked options.

unglue(**names*)

Remove each name from the list of glued options. If no list is given, remove all the glued options.

Filters are constructed with:

filter(**filters*)

Add each string argument as a filter. With no arguments, delete all existing filters. Note that if you attempt to filter using the name of an option for which you have not set a default using the facilities above, then any test in which the option is not specifically set will be not be executed.

Each `--filter` command-line option is simply a call to this function.

The command-line option –skip allows you to test your filters without executing any tests.

To assist you in constructing filters we have:

getOptions()

Return a dictionary of the options as they would be seen by a test defined at the location of this call. Intended to aide debugging of options.

filterdefs(*text=None*)

Add result of parsing text to the filter environment. Usually used to add functions to use in filters. If text is None, clear the environment.

Despite the power available here, we recommend you don't get too cute about it. The main thing is for it to be clear what is happening.

Customization

The Andyroid Tutorial contains ideas on various sorts of customization. These include defining your own postprocessor, main program, and application-specific input language extensions.

Using Levels

To use levels, make a master.ats file with stick commands separating the tests, such as this example input:

```
stick(level=10)
test("test1.py")
test("test2.py")

stick(level=20)
test("test3.py")
test("test4.py")
t5 = test("test5.py")

stick(level=30)
test("test6.p7")

# this test sets a level explicitly, that overrides the "stick".
testif(t5, "test7.py", level=10)
```

The currently "stuck" value is set in every test that does not explicitly set level. Thus test3, for example, has level 20, as if the level=20 were given in the test statement.

Executing ats on this file with the option –level 30 will execute all these tests. Executing ats with –level 15 will execute only test1 and test2; test7 depends on test5, which has level 20, so it will not be run even though it has level 10.

The Test Class

When a test is created by the test or testif command, a test object representing it is added to manager.testlist. This object is an instance of a class named `AtsTest`. Some users may wish to use the following details for debugging or postprocessors or customization.

The class `AtsTest` is available to users as `ats.AtsTest`.

`AtsTest(*args, **options):`

`stuck, glued, tacked`
: These are the current dictionaries for determining test options.

`test_number`
: The counter showing the number of tests defined so far.

`serialNumber`
: The unique serial number of this test.

`name`
: Set from an option to the test creation, or as the name of the script, or the name of the executable, plus the label. Eventually each test's name is made unique.

label

Set from an option to the test creation, incorporated in the name if given.

options

The options for this test, after resolution using defaults, stuck, tacked, and glued.

depends_on

If not None, the test instance this one depends upon.

dependents

A list of any direct dependents of this test.

exited

Has the job been run and exited?

output

A list of lines of magic output, newlines and magic removed

notes

List of notes from the run; user feel free to append to this list.

..attribute:: level

Test level set from resolved options. Same as `options.level`.

np

Number of processors required. Same as `options.np`.

batchDic

A dictionary that may contain various things for a batch job.

clas

A string containing the command line arguments after option interpolation.

executable

An Executable object specifying the executable's full path.

directory

The full path to the directory in which the test is executed.

groupNumber

The number of the group to which this test belongs, if positive.

groupSerialNumber

The number of the test within its group definition.

outname

The path to the standard output file for the test.

errname

The path to the standard error file for the test.

message

Explains the current value of `status`.

runOrder

A number indicating the order in which the interactive tests were run.

shortoutname

An abbreviated form of `outname` used for labeling.

timelimit

An object of class Duration – `timelimit.value` is the limit in seconds. Duration objects can be compared to integer numbers of seconds correctly.

waitUntil

A list of serial numbers of tests this one must wait for.

nosrun

Boolean value, runs the code without srun when `True`. This can also be used to circumvent the login node check so that a test can be run on a login node. When used it will be up to the test to get an allocation if needed.

set(*status*, *message*)

Set the object's status and message.

elapsedTime()

Returns a string, the formatted elapsed time of the run.

stick, unstick, glue, unglue, etc.

Class methods stick, unstick, glue, unglue, etc. are equivalent to the ones accessible in the vocabulary or ats module.

There are other methods that are not intended for end users.

Test Statuses

There are eleven status values that a test can have. This value is stored in the test's attribute `status`. Collectively this set of a statuses is in the list `ats.statuses` and each of them individually is in module `ats`.

Each status has a four-character abbreviation, shown in parentheses. The status can also be accessed under this name in the ats module. For example, PASS and PASSED are the same object. You can correctly compare two statuses using "is" or "is not", == or `!=`, or compare a status to a string representing its name or abbreviation, as in `PASSED == "PASS"`.

The statuses are:

INVALID (INVD)
The test was not properly stated. For example, it referred to a script file that did not exist. See the log file for the error.

CREATED (INIT)
The test was created but not (yet) run.

PASSED (PASS)
The test was run and succeeded.

FAILED (FAIL)
The test was run and failed.

EXPECTED (EXPT)
The test ran and failed in an expected way.

TIMEDOUT (TIME)
The test ran longer than its timelimit and was killed.

SKIPPED (SKIP)
The test was created successfully but skipped for some reason. The reason is in the test object's attribute `message`.

FILTERED (FILT)
The test was created successfully but filtered out for some reason. The reason is in the test object's attribute `message`.

BATCHED (BACH)
The test was deemed eligible for batch processing, and has been shipped off to the batch system. ATS does not know its fate.

RUNNING (EXEC)
The test is running, or was running when an error or keyboard interrupt occurred.

HALTED (HALT)
The test was stopped after running successfully for one minute. This status is only possible if the `--cutoff` command-line option is used.

Postprocessing

After ATS has finished executing tests, but before it exits, it calls any Python routines that have been registered with it by calling:

```
manager.onExit(routine)
```

The routine should have the signature

```
def routine (manager):
...
```

The routine can do anything it wants. In particular, manager.testlist is available. Here's an example of a trivial post-processor in an input file:

```
def routine(manager):
   passedTests = [test for test in manager.testlist \
           if test.status is manager.PASSED]
   print [test.name for test in passedTests]
manager.onExit(routine)
source ("set1.ats")
source ("set2.ats")
```

The postprocessing file is designed to make it possible to run postprocessing functions of this kind using the `state` variable as the `manager` argument, rather than doing it as an `onExit` routine.

Test Suite Strategies

One of the problems with excessive choice is the paralzying effect of choice. There are a lot ways to do things with ATS. So here we describe a basic strategy to use until you have enough experience to form your own opinion.

We strongly urge that you read the Andyroid Tutorial as well.

This scheme assumes your code sources are destributed over a set of directories with a common parent called Home, with a subdirectory Test.

In each subdirectory with code that has a separate test (such as a unit test, or a test that emphasizes that coding) put a file with extension "ats". This file contains a series of source statements that get further input or are test inputs containing introspective test statements).

::

test(clas = "-in myinput", np = 1) source("mysubdir/moretests.ats")

Separate these inputs into levels with stick-level statements such as:

```
stick(level = 10)
...some tests...
stick(level=20)
...longer-running tests...
stick(level=30)
...still more...
```

You choose how many different levels you like. We recommend choosing well-spaced numbers in case you later change your mind and want to insert levels between the ones you start out with. Note that any test can still specify a level on its own that would override the stuck level.

As you go up your directory tree toward Home, put files that source the ones below it, until finally you have a tree leading to a file, say "testsuite.ats", residing in your Home/Test directory.

Then you can make a series of small drivers. For example, your shortest test suite my be driven by this file:

```
glue("level <= 10")
source("testsuite.ats")
```

Running ats with this file as its input will result in only tests with level 10 or less being executed.

When the team that maintains a certain area wants to add a test, they add it to the closest member of the test-file tree relative to the source code they work with. They put it in the file at the appropriate level. This scheme leads to only rare source-code control conflicts, and ones that are usually a trivial merge; this avoids the conflicts generated by having a central test file.

Teams should be encourage to use introspection so that other members, less informed about how to test a certain area, can nevertheless exercise a good suite of tests using ATS, while allowing the experts to still use the input file directly with the code.

If there is one principle program being tested, it makes sense to use the -e option for it, and only explicitly specify an executable when it is different.

::

mycode = '/full/path/to/my/code' test(executable=mycode, script='foo.py')

The extended example in Examples/Andyroid gives you many more ideas about how to use ATS.

Porting and Custom Machines

ATS decides on which machine characteristics to use by examining the value of the environment variable MACHINE_TYPE; or, if it is not defined, the value of the environment variable SYS_TYPE; or as default the value of Python's `sys.platform` variable.

The reason for this three-level structure is to allow you to distinguish machine architectures when you have machines of the same basic type but with varied environments such as current OS level, parallel processing directives, or attached hardware. For an ordinary user on a personal computer, there is no reason to do anything special.

Most of the interaction between ATS and the platform takes place in a machine module, defined by default in the sources in file `Lib/machines.py`. Different behaviors are obtained by inheriting from this module, or one derived from it,

and overriding various methods. We then connect our new machine module to a value for MACHINE_TYPE with a comment in our module file, and install that module in a directory in the Python distribution.

Porting ATS to a new platform is just one of the things you can do with the technique we describe in this section; you can also do things like doing something special when a job finishes, inventing your own scheduling algorithm, etc. You'll need a decent knowledge of Python to do it, but you don't need to be an expert.

If you invent a new value for MACHINE_TYPE, you can change the way ATS launches and finishes jobs and keeps track of resources, amongst other things. You can add command-line options and react to the user's use of them. Your options will even appear when the user executes with `--help`.

To do this, you write a new Python source file, usually having a module name equal to your value for MACHINE_TYPE. This file must define a new child of `machines.Machine`, and you must have a comment:

```
#ATS:name module class npMax
```

This line or lines defines the relationship between a MACHINE_TYPE and this module's machine class and provides the maximum number of jobs you wish to execute at once (or it may mean the maximum number of processors one job can use in a parallel programming environment):

- name is the name to match with MACHINE_TYPE.
- module is the name of the module file, or SELF.
- class is the name of the class in that module to use as a Machine.
- npMax is a limit on np; if this number is negative it is a suggested default only.
- `machine.scheduler` is created by the standard `__init__` method of the machine. If you want to create your own scheduler you can replace this attribute. See *Customizing the Scheduler* below.

The file `Lib/machines.py` is well documented and it is usually not a large problem to get things working.

Once you have your module file ready, you write a setup.py file to go with it:

```
from distutils.core import setup
myMachines =[myMachine.py]   # list your machine module files
setup(name="myAtsAddon",
     author = "you",
     version = "1.0",
     description = "All About My Machine",
     data_files = [('atsMachines', myMachines)],
     scripts = ['mycustomdriver'],  #if you have one
)
```

and then execute `python setup.py install`. Set the environment variable MACHINE_TYPE and run ATS. It will report the machine module it has discovered.

In this `setup.py` file, the unchangeable word is `atsMachines`. This is the name of a directory below your Python installation root where the machine files are found by ATS. The scripts line can be omitted if you do not want to install your own driver.

Installing Machines as Plugins

As an alternative to the `data_files` approach you can register custom machines with ATS using setup tools' entry points plugin mechanism. This may be convenient if you are building more customized ATS wrappers that are themselves packages, but can also be used on standalone plugin packages. In this install method, ATS will look for these in the group `"ats.machines"`. This entry point name space is required for ATS to find your machine plugin. The example below shows how to set this up with a `pyproject.toml` build system using setuptools:

```
...

[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

...

[project.entry-points."ats.machines"]
custom_slurm_proc_sched = "mywrapper.atsMachines.
↪myAtsSlurmProcessorScheduled:MyAtsSlurmProcessorScheduled"
```

This shows adding a `custom_slurm_proc_sched` machine that's defined in the parent wrapper `mywrapper`'s atsMachines submodule, where the `MyAtsSlurmProcessorScheduled` class is defined in `myAtsSlurmProcessorScheduled.py`. This makes the machine name `custom_slurm_proc_sched` available to ats to use to instatiate a new machine. With this case you can either write whole new machines from scratch, or inherit from one of the default machines to change it's behavior. For more details see the setuptools documentations, which also includes more how-to's for `setup.py` and `setup.cfg` based packages.

Note one major difference with this method currently: the default machine config is not read from the `#ATS:name module class npMax` comment. The name, module and class gets read in from the plugin info, but the `npMax` field is not set. It defaults to -1 in this case; the current convention is to use env vars to override it inside the machines, so be sure and set those accordingly when configuring your custom machines.

Adding Test Options Via Machine

In a customized machine, the `examineOptions` routine can add entries to a dictionary, `options.testDefaults`. These will be default option values for each test. For example, here is how you would add an option `nt` that could be specified on the command line in the machine file:

```
def addOptions(parser):
    parser.add_option('--nt', dest='nt', default=1, type='int',
        help='Set default number of threads per test.')

def examineOptions(options):
   options.testDefaults['nt'] = options['nt']
```

Of course, the machine would also have to examine and use properly the value of each test's option `nt`; but it would always have one, and hence it could be used in filters.

Customizing the Scheduler

The scheduler class StandardScheduler is defined in module `schedulers`. It handles issues such as priorities, and enforcing rules for the `group()` and `wait()` commands, and the `independent` option.

Customizing the scheduler is possible but difficult. It should in particular supply a method testlist() that returns the list of tests that are not yet completed. Inheritance is strongly suggested, so that you only change what you need to change. You'll probably want to change the machine too so that it creates the correct scheduler, but it feasible to create and assign a new machine attribute `scheduler` at any point up to and including the call to `machine.load`.

The important thing is to maintain correct separation between the scheduler and the machine objects. The scheduler must ask the machine for such things as `canRunNow` that are within the purview of the machine, and ask it about whether jobs have finished. The machine contains an attribute `running`, a list of the jobs currently running. The `periodicReport` in the scheduler does the basic report once a minute; a machine can call this and then add more.

The ats Module

The `ats` module can be imported in custom drivers and postprocessors. Resources available in it are all imported from internal modules. These are documented further in the Appendix.

log, terminal

See the discussion of the log. `terminal` is simply a version of `log` that only writes to the terminal, not the log.

times

Is a module containing useful time-handling routines

configuration

Is the module that has information about the machine and command-line options.

manager

Is the manager object. It has in particular `testlist`, and the routines discussed above. It is defined in the `management` module.

testEnvironment

Is the vocabulary dictionary.

AtsTest

Is the test class.

debug(*value=None*)

Is the debug function

exception AtsError

Is the class of exceptions thrown by ATS.

statuses, CREATED, INVALID, PASSED, FAILED, HALTED, SKIPPED,
BATCHED, RUNNING, FILTERED, TIMEDOUT, SYS_TYPE, MACHINE_TYPE

Discussed previously, these are available via the ats module as attributes.

Using A Batch Facility

General Information

When running ATS, if a batch facility exists, both the interactive jobs and batch jobs will run. You have to use the facilities of that batch facility to find out what happened to those tests, because ATS will likely finish and exit long before those jobs are done.

Unfortunately, the world doesn't have a standard batch facility. So here is an example of using the MSUB batch system at the Livermore Computing Center. Much of what follows would apply to any batch system.

To add a different batch system one must customize a batch machine to be installed in your ATS. For advice on how to do this, please contact us.

The basics are simple: if a test has a `batch = 1` option, it is a batch test. Each of the batch tests are individually submitted to the batch system. The `--allInteractive` flag is available to execute such tests without using the batch system. Otherwise, they are simply skipped if no batch system is found.

For the LC system in particular,

- A *testName*.bat file is craated for the test.
- The test information is written to a "batchContinue.log". This file will be a concatenation of all the batch tests and will provide information about the tests.

Running Entirely In Batch

Submitting a lot of single batch jobs may overwhelm some batch systems. In that case it may be preferable to submit just one big batch job. One batch job is created to run all the tests (both batch and interactive).

The ATS option `--allInteractive` is neccesary in the ATS command to prevent the tests from being submitted seperately as batch.

An example of a batch script using MSUB at LC:

```
#!/bin/csh

#MSUB -N tmpAts0.157456004499.job
#MSUB -j oe
#MSUB -o tmpAts0.157456004499.job.out
#MSUB -q pbatch
#MSUB -l nodes=4:ppn=16
#MSUB -l ttc=64
#MSUB -l walltime=200
#MSUB -V                    # exports all environment var
#MSUB -A myBank             # bank to use

setenv SYS_TYPE chaos_4_x86_64_ib

date
cd /my/work/directory/; atsb --allInteractive --numNodes=4  -useSrunStep Test/full.ats
date
```

The command-line options `--numNodes=4 --useSrunStep` are not a part of standard ATS. In this case, the ATS machine type `chaos_4_x86_64_ib` has been defined in a custom machine file, and custom machine files can add command-line options.

More Examples

Introspection

```
mytestA.py:
  #ATS:test(SELF, batch=1, np=2, ...)
  ...mytestA problem...

mytestB.py:
  #ATS:stick(batch=1)
  #ATS:test(SELF, ...)
  ...mytestB problem...

myAts.ats:
  tack(batch=1)
  source('mytestC.py')
  source('mytestD.py')
  source('mytestE.py')
```

In `myytestA.py`, a 2-processor batch job is created by introspection.

In `mytestB.py`, the test created through introspection will be run in batch, unless it happened to explicitly contain the option batch = 0, because the `stick` call makes batch = 1 the default in this file.

Running `myAts.ats`, the `tack` makes batch = 1 apply also in the three files that get read. If this were a `stick`, it wouldn't apply inside those other files.

Test Control

Suppose the file mytest.py contains a test script. The script throws an exception if it gets an error. It has a command line argument delta. Suppose mytest.py reads:

```
#ATS:log('mytest.py tests sanity of my group leader.')
#ATS:test(SELF, 'delta=0.5')
#ATS:test(SELF, 'delta=0.6', sanitycheck = 1)
#ATS:test(SELF, 'delta=0.7', np=4, sanitycheck = 1)
import physics
...command line processing to get delta's value...
...test problem....
...throws an exception if it fails...
```

If we run:

```
ats --exec myapplication mytest.py
```

then it is equivalent to running 3 tests:

```
myapplication mytest.py delta=0.5
myapplication mytest.py delta=0.6
myapplication mytest.py delta=0.7
```

The last one is run on 4 processors if the machine supports it.

Consider the command line:

```
ats --exec myapplication -f 'sanitycheck == 0' mytest.py
```

None of the tests are run; the first because sanitycheck is not one of its options, the other two because it is but the value is not zero. We could make sanitycheck have a default value of zero for all tests in mytest.py by adding this line to the top of mytest.py:

```
#ATS:stick(sanitycheck=0)
```

With this line added we would run only the first test.

Using the filter sanitycheck==1 would run the last two tests but skip the first. Using the filter 'not np' would run only the first two jobs, since they have by default np == 0.

Suppose mytest.ats reads:

```
source('mytestA.py')
source('mytestB.py')
```

and mytestA.py reads:

```
#ATS:stick(batch=1)
#ATS:test(SELF,delta=0.1)
...mytestA problem....
```

and mytestB.py reads:

```
#ATS:test(SELF)
...mytestB problem....
```

If we run:

```
ats -e myapplication --nobatch mytest.ats
```

then only myTestB.py is executed, and execution of mytestA.py is skipped, since ats is not set for batch tests to run. Note `--exec` can be abbreviated as `-e`.

If we run:

```
ats -e myapplication mytest.ats
```

then mytestA.py is submitted to batch and mytestB.py is run interactively. If there is no batch system, mytestA.py is skipped.

In practice a batch facility, if present, would add further options for controlling itself, such as options to set accounts or priorities or timelimits. The maintainers of such batch facilities will provide the documentation for them.

Finally,

```
ats --allInteractive -e myapplication mytest.ats
```

will test both myTestA and myTestB.

Resources For Learning ATS

The *Examples* directory in the distribution contains the sources that accompany the Andyroid Tutorial, including some sample customizations.

The *Test* directory contains more examples, although care must be taken in reading them as some of these are designed to fail.

At your particular location you may find other directories that define machines and batch systems for your local computer center.

Quick Recipes

- To run only the batch tests:

```
ats --filter 'batch == 1'  mytest.ats
```

- To run only the interactive tests:

```
ats --nobatch  mytest.ats
```

- To run all tests as interactive tests:

```
ats --allInteractive mytest.ats
```

- To check your input add –skip; add –debug for even more information.
- To keep the output files even if the test succeeds, add –keep

1.4.3 Notes

This chapter contains documentation useful for maintainence, customization, and debugging.

Modules

The ats module contains several submodules documented below. The `ats` program imports the `ats` module and calls the manager's `main` routine. As documented in *Custom Drivers*, a user may create their own driver and even break `main` down into pieces in that driver.

ats

configuration

The configuration module makes the basic discoveries about the machines, creates the log, requests command-line options from the machines, and processes the options with call-backs to interested parties to examine them.

management

The management module is the main supervisor of the program, and is instantiated as a singleton object, `manager`.

tests

This module defines test objects and groups. However, these are not created directly but rather via functions in the manager, `test`, `testif`, `group`, `endgroup`.

schedulers

The scheduler attribute of the machine is an instance of the `StandardScheduler` class.

machines

(See also *Porting*.)

This module contains base definitions for interactive and batch facilities. To adapt to a new platform, inherit from machine and override appropriate methods.

log

The log is an instance of `AtsLog`. The log object is callable (See the `AtsLog.__call__` method). A call is equivalent to the method `write`. The log call can write to a file, the terminal, or both.

An instance of AtsLog named `terminal` is also available. This writes only to the standard out, not to any file.

times

This module contains utility functions and a class that deal with times.

atsut

This module contains utilities and definitions (such as the statuses) used widely in ATS. The basic error type `AtsError` is also defined here. Many of these definitions are imported into the `ats` module proper. The class AttributeDict is used in several places. It is a dictionary that also accepts attribute-style reading and writing.

executables

This small module is used to represent executables.

Programming Notes

Note that because of the complex interactions between priorities, dependents, filters, and waits, the `AtsTest` and `AtsTestGroup` classes cannot be directly instantiated by a user. The purpose of making those classes visible at the ats module level is to allow subclassing.

Forming Groups

Each test has a `group` attribute. These are instances of *AtsTestGroup*. Under normal circumstances each test gets a new group instance with a distinct group number, and that test is the only method of that group. Doing this avoids a considerable amount of logic compared to only having groups for tests created in the scope of a *group()* command.

When a *group()* call occurs, the *newGroup* class method of the *AtsTest* class is called. This halts the incrementing of the group number and subsequent tests that are created share the group instance until either `endgroup()` is called and calls the class method `endGroup`, or we reach the end of the source file, which triggers a call to `endGroup`.

Note that a `group` call can specify keyword / value pairs which bind more tightly than anything except an explicit pair in a *test* statement. This allows the user for example to specify a base label, with the other members of the group getting the same name with a `#n` numbering by default.

The group objects inherit from list and are basically a list of test objects with routines added to treat the list as a collection.

Implementing Waits

Three *AtsTest* class methods combine to implement *wait()*: *waitNewSource*, called when a new file is begun; *waitEndSource*, called at the end of a sourced file; and *wait* itself, called by the user.

The result is that each test object ends up with an attribute *waitUntil* which is a list of the tests this object must wait for. Note that this attibute (on the test object, not the one on the class) must never be modified because it may be shared with another test. You will note in the coding several instances of such lists being copied with a colon selector, in order to avoid unwanted sharing.

Since many of these lists are long stretches of consecutive integers, it would be possible to save space by making them instances of a special class that acts like a list. We have not yet done this and will until users decide they are happy with the semantics we have currently implemented.

Dependents

Each test has a list of all of its direct and indirect dependents. These lists are created via the method `addDependent` of `AtsTest` called by the `testif` function.

This method enforces several important policies, such as disabling tests that are children of tests that will never run or which are expected to give a failing result, or which are to be batched.

The need to enforce these policies drives the decision to do *canRun* early. This means that by the time a dependent is created, the status of its parent(s) has been fixed as to filtered, skipped, or batched. Note particularly the case where an otherwise interactive test gets switched to batch because it cannot run on this interactive machine.

The Standard Machine

As tests are created, the `canRun` method of the interactive machine is called to determine if a test can run when the machine is empty. Assuming a test makes it into the final interactive test list, all of which are in status CREATED, we need to decide the order in which the tests are to be run.

This order is dynamic, as it depends on processor availablity. Other factors are the results of `wait` and `group` commands.

There are four conditions that must be met to run a test:

1. The test has status CREATED.
2. Enough processors are available.
3. The directory where the test is to be executed is not "blocked". The test would not be affected if its option `independent` is True. Otherwise there must not be a non-independent test or group currently reserving that directory (that is, another test is running there or a group was started there that isn't finished yet).
4. Any parent tests are finished and have passed, and any tests this one must wait for because of `wait()` calls are no longer waiting to run.

As tests complete, any failure may put descendents into SKIP status.

During the `load` of the interactive test list, the `totalPriority` of a test is calculated using the test's list of children and tests that must wait for it. The sum of the priorities of such subordinate tests becomes the `totalPriority` of the test. The test list is then sorted on `totalPriority`.

To choose the next test to start, then, we take the first test in the list that satisfies the four conditions. (The routine `canRunNow` tests this.)

As tests complete, we must eventually find a new test to run if there is one whose status is still CREATED, because when no test is running any more, no directory is blocked and the tests have all been certified runnable on an empty machine by `canRun`.

When we can't find such a test, we' re done!

INDEX

Symbols

A

B

C

D

E

F

G

P

R

S

T

U

V

W