Pandokia

Release 1.5.2rc2.dev2+g555b2fff

Mark Sienkiewicz and Victoria Laidler

Mar 02, 2021

Contents

CHAPTER 1

Overview

1.1 Philosophy

You don't run tests because it is virtuous; you run tests because you need to see the results from those tests. If you have many tests, the results can be difficult to manage.

Pandokia is a system for displaying those test results.

The two major subsystems are:

- A web-based user interface for reporting. It uses a CGI, so there is no complicated server configuration.
- An optional system for executing tests. It can run tests written for many different test frameworks, all in a single test run. There is direct support for tests in py.test (python), nose (python), unittest2 (python), shunit2 (sh), FCTX (C/C++), or even stand-alone programs. You can add your own methods for executing other types of tests.

  You don't need to use pandokia to run your tests. The database ingest reads a simple text format; anything that can produce pandokia-formatted data can report test results. There is also an experimental interface for reading JUnit/XML data, but the pandokia format contains fields that JUnit does not report.

We assume two primary use cases for Pandokia:

- continuous-integration batch runs of the whole system
- developer-driven manual runs of parts of the system

The reports understand that you will run the same test code many times. Each test result is identified by a unique tuple:

- test_run is the name of this instance of running a batch of tests. The name is arbitrary, but often fits a pattern such as "daily_2010-04-12" or "user_henry_ticket87".
- project is a project name; a CI system may integrate tests from several projects into a single report.
- context identifies something about the environment where the test was run. We sometimes name contexts for the version of python the test ran in, or for a particular system configuration.
- host is which computer the test ran on. Our CI system runs the same tests on many different operating systems.

- test_name is the name of a particular test. The reporting system understands that test names fit into a hierarchy, much like files are organized in directories. Test names are arbitrary, but the test runners that Pandokia provides will name the test by the directory where the test file is located, the name of the file containing the test, and the name of a function/class that implements the test.

  The system assumes that you always use the same name for the same test, so that you can compare results across test runs, hosts, or contexts.

After you ingest the test results into the database, there are reports that cover:

- all currently known test runs
- a tabular report of all information about a single test run, or a project within that test run
- a tree view of the results, with "drill down"
- a summary of an arbitrary list of tests, with operators to add/remove tests from the list and combine lists
- details of a specific test, including captured stdout/stderr and various values ("attributes") that the test reported

1.2 Test status

We find it useful to have more to a test status than just "pass" or "fail". Pandokia presently supports the following statuses:

- P = Passed
- F = Failed
- E = Error: there was a problem running the test
- D = Disabled: test was directed to not run
- M = Missing: test was expected to run, but no report was received

Tests can be disabled by developers. This can be useful for chronically failing tests, and for tests that were written and committed before the code to pass them was written, as in test-driven development.

A table of expected tests (by identifying tuple) is kept in the database. Any tests that were expected but not received will be marked missing in the database. The usual model is to run a set of tests, import the results, then declare that those tests are all expected; the result is that a new test can become "expected" simply because it once reported a result.

1.3 Test attributes

Pandokia can collect additional information beyond simple Pass/Fail by declaring "attributes".

Test definition attributes (TDAs) can be used to record input parameters or other information about the definition of a test such as reference values.

Test result attributes (TRAs) can be used to record more detailed results, such as computed values and magnitude of discrepancies.

The browsable report generator will optionally show these values for a selected set of tests, and will pull out all identical values for a given attribute into a table near the top of the report. This makes it easy to quickly identify what failing tests may have in common.

1.4 Browsable reports

The browsable report generator allows the user to navigate through the database of test results in tabular or tree form, gradually narrowing the test set of interest down to the level of a single test report, if desired. The figure illustrates the navigation paths supported by the interface.

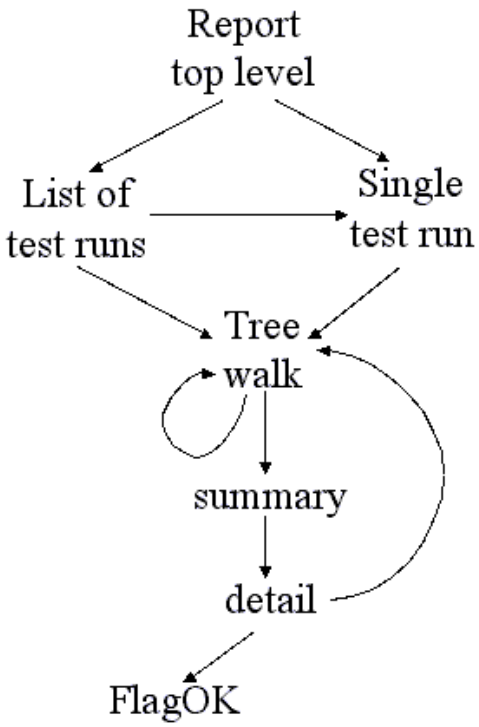

Fig. 1: This figure shows the navigable flow through the browser report generator.

The top level presents you with a list of test runs that are available to browse through, and also the option to specify a given test run (for example, the daily_latest run).

Either path will take you to the treewalker, with which you can navigate through the various test subsets, both hierarchically through the test namespace, and by project, host, test run, and status.

Once you've narrowed to a subset of interest, the "show all" link will take you to a summary report for this set. The summary report presents a table with one row per test, including contact and status. The summary report permits the comparison of this subset to the same subset in another test; displaying test attributes; and sorting by table column.

Clicking on a test name takes you to the detailed report for a single test, which contains all the information about the test available in the database. For tests that are "OK-aware", a "FlagOK" button is present on this form, that can be used to mark a failed test "OK".

From the detailed report, you can return to the treewalker.

The following screen shots illustrate several commonly-used reports in the system:

- A `daily report` in tabular form
- An `intermediate report` from a tree-navigation
- A `single test result`
- A set of failing tests that have `attributes in common`

1.5 Internal workflow

A high-level view of the system is quite simple, as illustrated in figure: running tests produces a log file containing test result data, usually with many test results appended in a single file. An importer processes the log file and loads the data into the database. The accumulated results are then available to users through a CGI browser interface.

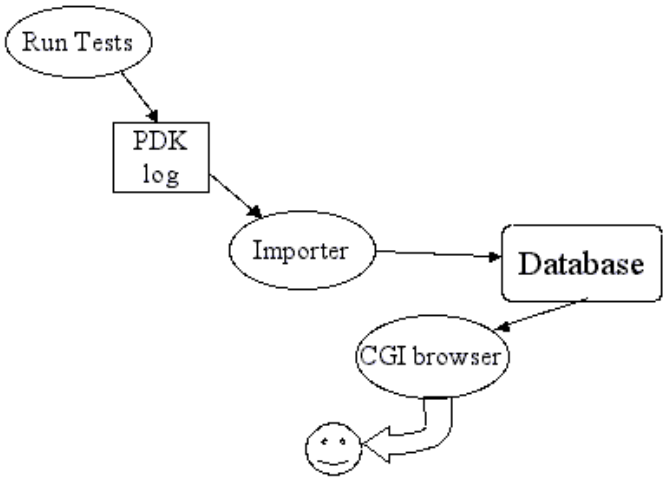

Fig. 2: High-level view of system dataflow.

This section discusses the internal workings of the system, as illustrated in the more complex diagram below.

In addition to the standard test-import-browse data flow, some additional flows provide enhanced bookkeeping.

- Multiple contacts can be associated with sets of tests; this information may change, and changes must be imported.
- A notifier sends a customized email with reports of any anomalous (fail, error, or missing) test results.

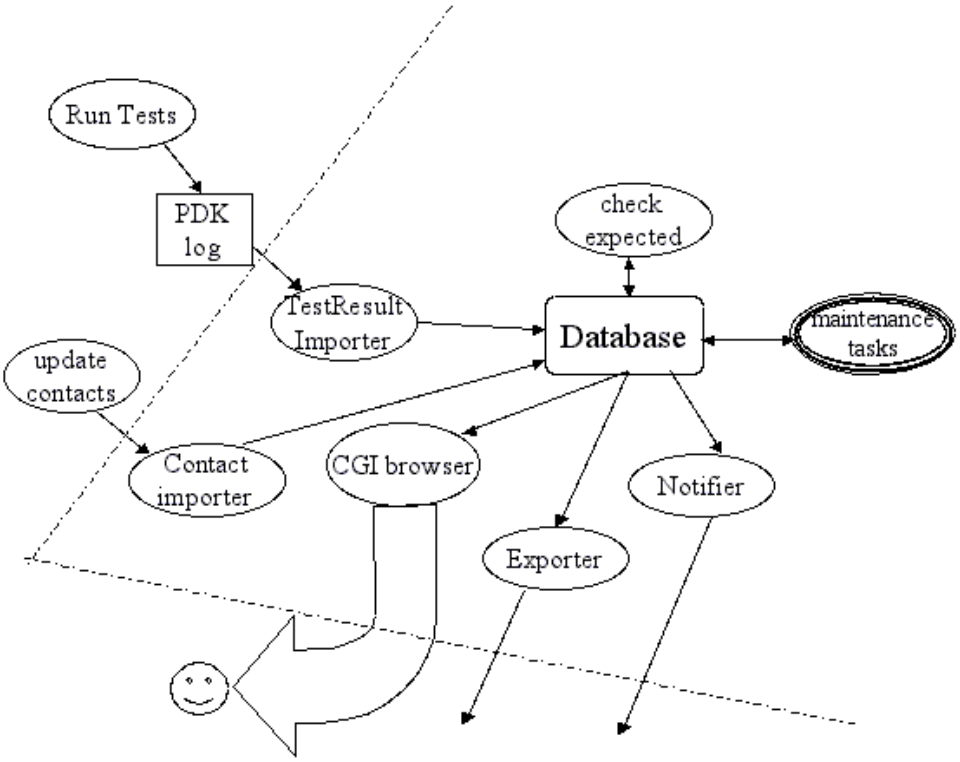

Fig. 3: This figure shows the elements of Pandokia in more detail. The dashed line marks the boundary of the server that hosts the web interface.

- Missing tests are detected by checking against a list of expected tests, which is automatically updated when new tests are added.
- Test reports can be exported from the database, and a small set of database maintenance utilities are provided.

Test discovery is performed hierarchically in a directory tree. Each directory may contain special files specifying the environment, contact information, or filename patterns; this information is applied hierarchically, so that the configuration in a parent directory applies to its children unless the children override it. A test named testfoo.py may be disabled by placing a file in the same directory named testfoo.disable. The test discoverer will not pass this file to any test runner.

The test meta-runner invokes the correct test runner within an appropriately configured environment for each test file found (locally, we use nose and a home-grown system). When processing a directory tree, multiple test runners can be invoked concurrently, but only one test runner at a time will be invoked per directory. When running multiple tests concurrently, the system creates multiple log files. All the log files are imported into the database to obtain the full results.

The importer processes a test result file and uses the information in it to update the various database tables. The missing test identifier then compares the tests found in a given run against the set of expected tests, and inserts records for any missing tests with a status of missing. If a test report is imported for a test previously considered missing, the database will be updated accordingly.

The reporter provides a browsable interface to several reports, organized by lists or by trees. The user can examine attributes for a group of tests, compare results to a previous test run, or click through to an individual test report.

1.6 Authors

Mark Sienkiewicz (STScI) and Vicki Laidler (CSC/STScI), Science Software Branch, Space Telescope Science Institute

1.7 Support

You can request help on Pandokia by sending email to help@stsci.edu with SSB/Pandokia in the subject line. The authors also follow the TIP mailing list (testing-in-python@lists.idyll.org).

CHAPTER 2

Installing Pandokia

Pandokia works on Unix/Linux and Macintosh machines. The procedure is the same in all cases. There has been some effort toward Windows support, but it is incomplete.

The sequence is to first install the software, then configure the resulting system.

2.1 Initial Install

Download either the tar or zip file from http://ssb.stsci.edu/testing/pandokia/

Extract the file:

```
tar xf pandokia-1.2.tar.gz
```

```
unzip pandokia-1.2.zip
```

Install the software:

```
cd pandokia-1.2
python setup.py install
```

If you want to install it somewhere other than the default python site-packages, you can use regular setup.py options. For example:

```
cd pandokia-1.2
python setup.py install --home $HOME/my_python
```

In case you chose a non-standard location for your install, the setup.py will print example shell commands that you can use to set up your environment. If you did not, you can ignore that output.

2.2 Web Application

To install the web application, you need to:

- install the CGI
- configure and initialize the database

2.2.1 The CGI

You can install Pandokia in any web server that can run CGI scripts. We use Apache for our normal usage.

When you run setup.py, one of the things it will say is something like:

```
Get the CGI from  /Users/sienkiew/test_install/python/bin/pdk
```

Copy that file into the appropriate place in the DocumentRooot for your web site, so that your web server can run it as a CGI. If your server allows symlinks (Options FollowSymLinks in Apache), you can use a symlink:

```
cd /var/www/html/cgi-bin
ln -s /Users/sienkiew/test_install/python/bin/pdk pandokia.cgi
```

Otherwise, you can copy the file:

```
cd /var/www/html/cgi-bin
cp /Users/sienkiew/test_install/python/bin/pdk pandokia.cgi
```

This file is the only part of Pandokia that must be in the DocumentRoot of the web server, but it requires the rest of Pandokia to be installed on the system. You cannot just copy *pdk* to another machine.

Once you have done this, the pandokia application will be present on your web server. In the example above, the URL would be something like http://www.example.com/cgi-bin/pandokia.cgi

The Development Web Server

Pandokia comes with a mini development web server that you can use for testing. You can run it from the *bin* directory where Pandokia commands were installed. For example, if the setup.py said:

```
scripts went to /Users/sienkiew/mypython/python/bin
```

Then you could start the development web server with:

```
cd /Users/sienkiew/mypython/python/bin
cp pdk pdk.cgi
pdk webserver
```

It will state the IP address and port number that it is listening at. Load that page and click on pdk.cgi.

We do not recommend using this web server for serious use, but it is good enough for preliminary tests.

2.2.2 The Database

Initializing the database : sqlite

The advantages of Sqlite are:

- It does not need a database server.
- It is usually already built in to python.
- It very easy to create a new database.

The disadvantage is:

- It has very coarse write locking. If you run a big transaction, other users may get timeout errors.

Sqlite3 support is normally built into Python. You can look for it by

```
python
import sqlite3
```

If there is no error, you already have sqlite support. If you do not have sqlite, you can re-compile python with sqlite support, or you can install pysqlite from http://pypi.python.org/pypi/pysqlite/

Get the name of the Pandokia configuration file by entering the command:

```
pdk config
```

Edit that file. In the section marked "# Database: SQLITE", change "if 0 :" to "if 1 :" and set the value of *db_arg* to the name of the database file.

The database file and the directory that it is in must be writable to

- the user that runs the CGI (e.g. "apache")
- the database administrator
- any user that imports data with "pdk import"

Apparently, some NFS servers still have buggy file locking, which you can avoid by storing the data files locally.

Create the database tables and indexes with the command:

```
pdk sql sqlite.sql
```

Pandokia uses "pragma synchronous = normal;" for speed. Certain types of crashes can cause your database to be corrupted. See http://sqlite.org/pragma.html#pragma_synchronous for more information. Pandokia does not have a configuration to change this, but you can change the setting in pandokia/db_sqlite.py

Initializing the database : MySQL

MySQL provides good performance. The only significant disadvantage is that you need to know (or find someone who knows) how to do MySQL database administration. This is mostly only an issue for the initial setup.

To use MySQL, the machine hosting your web server will need the MySQL client libraries and the MySQLdb python package. You need to have a machine running a MySQL database server, but it does not need to be the same machine as your web server.

MySQL is available from http://www.mysql.com/

MySQLdb (also known as "MySQL for Python") is available from http://sourceforge.net/projects/mysql-python/ ; we are using version 1.2.3 at STScI.

Create the database and a database user for the pandokia application. Pandokia needs a database user with the permissions USAGE, SELECT, INSERT, UPDATE, DELETE, and CREATE TEMPORARY TABLES.

Here is what SHOW GRANTS says for our pandokia user:

```
GRANT USAGE ON *.* TO 'pandokia'@'%.stsci.edu' IDENTIFIED BY PASSWORD 'XXXXX'
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE TEMPORARY TABLES, SHOW VIEW ON␣
↪`pandokia`.* TO 'pandokia'@'%.stsci.edu'
```

Get the name of the Pandokia configuration file by entering the command:

```
pdk config
```

Edit that file. In the section marked "# Database: MySQL", change "if 0 :" to "if 1 :" and set the values in *db_arg* to the access credentials. *host* is the machine that runs the database server. *user* is the user name to use to log into the database. *passwd* is the password to use to log in to the database, *db* is the name of the database.

You can use the readpass() function to store the password in a file or you can just write the password in the config file as a string literal.

Create the database tables and indexes with the command:

```
pdk sql mysql.sql
```

Initializing the database : Postgres

Postgres provides good performance. The only significant disadvantage is that you need to know (or find someone who knows) how to do Postgres database administration. This is mostly only an issue for the initial setup.

To use Postgres, the machine hosting your web server will need the Postgres client libraries and the psycopg2 python package. You need to have a machine running a Postgres database server, but it does not need to be the same machine as your web server.

Postgres is available from http://www.postgresql.org/

pyscopg2 is available from http://initd.org/psycopg/ or http://pypi.python.org/pypi/psycopg2

TODO: describe using postgres - this is roughly the same as MySQL. See the comment at the bottom of pandokia/db_psycopg2.py for some notes on using postgres .

```
pdk sql postgres.sql
```

2.3 Machines that will run tests

On a machine that will only use Pandokia to run tests, you do not need to make any configuration changes. You may find it convenient to install some supporting test frameworks.

None of this support software is required to install Pandokia. You can install Pandokia without any of this, then add it later.

- nose (Python) - http://readthedocs.org/docs/nose/
- py.test (Python) - http://pytest.org/
- unittest2 (Python) - http://pypi.python.org/pypi/unittest2
- fctx (C, C++) - http://fctx.wildbearsoftware.com/

  All necessary parts of fctx are included in pandokia, so it is not necessary to install anything to use fctx.

  Their web server has been down for a while. You can find a copy of the fctx documentation at http://ssb.stsci.edu/testing/fctx/

- pyraf (IRAF) - http://www.stsci.edu/institute/software_hardware/pyraf

  pyraf is used only to run IRAF tasks in the stsci_regtest runner.
- shunit2 (sh) - specially modified version from http://ssb.stsci.edu/testing/shunit2/

CHAPTER 3

Using Pandokia with Existing Tests

3.1 Step One: Just use it

If you already have existing python tests that can be recognized and run by nose, then you can take advantage of Pandokia's reporting features as soon as you have installed and configured it.

Place a file named pandokia_top at the top of your test tree, and then issue the command:

```
pdk run -r <dirname>
```

This will recursively search through the directory, discovering tests and running them with nose, and using the included nose plugin to write the test results to a log file. Default values for the project (taken from the current directory) and log file (taken from the user and timestamp) will be used, and printed to stdout. Then:

```
pdk import <logfilename>
```

will import the results into the Pandokia database, and then you can use the browsable report generator to examine your reports.

3.2 Step Two: Environment, notification, and disabling

To take advantage of more of Pandokia's features, add some special files to your test tree.

3.2.1 Customizing the environment

A file named pdk_environment can be placed in test directories to define the environment in which your tests will run.

A [default] section defines environment variables applicable to all test environments, and optional named [osver], [mach], and [hostname] sections can customize the environment to be used when the tests are run on different machines. These customizations will override the values in the default section.

For example, suppose Arthur has a set of tests that require a resource that is located in different places on different machines. Then his pdk_environment file would look something like this:

```
[default]
PDK_PROJECT = odyssey
monolith = black
doors = $podbay/doors/open.com

[hostname=hal]
podbay = /data/discovery/podbay

[hostname=mycroft]
podbay = /usr/local/moonbase/podbay
```

Now the tests that run on *hal* will run in an environment that includes the variables:

```
PDK_PROJECT = odyssey
monolith = black
podbay = /data/discovery/podbay
doors = /data/discovery/podbay/doors/open.com
```

while the tests that run on *mycroft* will run in an environment that includes the variables:

```
PDK_PROJECT = odyssey
monolith = black
podbay = /usr/local/moonbase/podbay
doors = /usr/local/moonbase/podbay/doors/open.com
```

Note that the default section may refer to variables defined in the custom sections.

Environment files are applied hierarchically. Suppose Arthur has a test directory tree laid out like this:

```
odyssey
    earth
    moon
    jupiter
```

Arthur can place the file described above in the odyssey directory, and its values will apply to all tests in the tree. But he can place an additional pdk_environment file in odyssey/jupiter that contains only the following:

```
[default]
doors=$podbay/doors/closed.com
```

and this value will override the value of $doors for tests in the jupiter directory only (and any subdirectories it may have).

3.2.2 Disabling tests

The presence of a file named sometest.disable will prevent a corresponding sometest.py file in the same directory from being examined for tests. This can be useful to disable chronically failing tests (although this feature should be used with caution!), or in test-driven development mode, to disable tests that you know will fail because you haven't fixed the bug or written the code for them yet.

You can also disable a test for just one context by creating a *.CONTEXT.disable file. For example, if you want to disable a test named foo *only* in the orange context, then you would create a file named foo.orange.disable in the same directory as foo.

3.2.3 Enabling tests

Support for enable files was added to compliment the disable functionality. The presense of an enable file supercedes any disable files for a particular test. The enable files work just like disable files: for some test foo, you can have foo.enable which means that it *always* runs, or you can have foo.whatever.enable, which means that foo only runs when $PDK_CONTEXT has the value of "whatever". If foo.enable or foo.*.enable exists, all foo.disable and foo.*.disable files are ignored by Pandokia.

3.2.4 Email notifications

You can use Pandokia to send customized email notifications of failed, error, or disabled tests. A special file named pdk_contacts can be placed in each test directory. This file should contain the usernames or email addresses, one per line, of people who should be notified when tests in this directory fail.

The contact files are applied hierarchically but cumulatively. For example, consider the following directory layout with pdk_contacts files populated as follows:

```
film/pdk_contacts: stanley
   odyssey/pdk_contacts: arthur
      earth
      moon
      jupiter/pdk_contacts: hal
   clockwork/pdk_contacts: anthony
```

Then Stanley will receive an email containing information about all failed tests; Arthur will receive email containing information about the odyssey project, including all of its subprojects, and Anthony will receive email about only the clockwork project. Hal will receive mail only about the jupiter subproject of odyssey.

Unlike the previous two features, you will have to issue a couple of commands in order to update the contact fields in the database. On the test machine, run:

```
pdk_gen_contact projectname /directory/name > contact_list.txt
```

will construct a table of contact information from the contact files in your test tree. "projectname" is the name of the project that you are processing and /directory/name is the path to the root of a test tree. (That is the directory that has the file pandokia_top in it.)

The assumption is that the project contains the same set of tests on all machines, so it is only necessary (or even useful) to run pdk_gen_contact on a single machine.

(b.t.w. pdk_gen_contact is a hack; it will be replaced by "pdk gen_contact" in a future release.)

On the server machine,

```
pdk import_contact < contact_list.txt
```

will update the database with that information, so it will be available when you run:

```
pdk notify
```

after importing the results of a test run.

The contact names are also shown in the summary report available in the browsable report, which can be useful information for a release manager or someone assigning help desk calls.

3.3 Step 3: Add attributes to your tests

Pandokia can collect and report additional information about your tests through the population of test attributes.

Test definition attributes (TDAs) are typically populated by values that are known when the test is being written. It can be used to record input parameters, reference values, thresholds, or descriptive information about the test that would be useful to have when analyzing failures.

Test result attributes (TRAs) can be used to record more detailed results than a simple pass/fail status, such as computed values and discrepancies. They are typically populated by values that are computed during the execution of the test.

Making use of test attributes requires modifying your tests. For tests that inherit from unittest.TestCase, you can add:

```
self.tda=dict()
self.tra=dict()
```

to the setUp() method of your test class; then populate the dictionaries as desired.

For test functions, declare the dictionaries as global variables in your function, then populate them (but do not redefine them) in your tests:

```
tda=dict()
tra=dict()

def test1():
    tda['year']=2001
```

The Pandokia plugin will take care of clearing the dictionary contents between tests to avoid cross-test contamination.

Additional examples of how to add attributes to your test cases and functions can be seen in example_test_case.py and example_test_function.py

3.4 Step 4: Use helper functions to write new tests

Pandokia provides some helper classes and functions to facilitate writing some kinds of tests.

3.4.1 File comparisons

The file doc/example_filetest.py contains examples of how to subclass from and use the FileTestCase class in your own tests. This class pre-defines several methods:

- command() executes a shell command
- check_file() compares a file to a reference file
- tearDown() cleans up any compared files for tests that passed.

Both the .command() and .check_file() methods automatically populate the tda and tra dictionaries with useful values.

These methods use the helper functions in helpers/filecompare.py, which can also be called independently.

3.4.2 Functions with attributes

Adding TDAs and TRAs to test functions can be done by implementing the dictionaries as global variables. (See example_testfunction.py for an example)

Alternatively, a developer can inherit from the FunctionHolder, and write methods for it as if they were functions. This class pre-defines the tda and tra dictionaries in its setUp.

3.5 ok-ify tests

section here about the TDA _okfile and what to put in an okfile

CHAPTER 4

Using Pandokia To Run Tests

abstract This describes how to use the Pandokia Meta-Runner to execute your tests.

The Meta-Runner identifies tests, sets up an appropriate execution environment, and invokes a Test-Runner to actually performs the tests. If parallelism is appropriate for your tests, you can direct it to run multiple tests concurrently.

The Meta-Runner has provisions for enabling/disabling tests and for setting environment variables before the test is executed.

A Test-Runner is a Pandokia component that implements a standard interface between the Meta-Runner and some specific test execution software. There are several test runners available. For example, the "pytest" Test-Runner uses py.test 2.2.1 (along with a plugin) to run tests.

See adding_runners.rst for documentation on implementing your own Test-Runner for whatever testing systems you have.

Contents

4.1 Simple Overview

Typical usage is to create a directory tree that contains all the tests. Every test has a unique name, and the path from the top of the directory tree to the file containing the test will be part of the name.

There are three ways to run tests with the Pandokia Meta-Runner:

- You can recursively run all the tests discovered in a directory tree

```
pdk run -r directory

pdk run --recursive directory
```

- If you want to run all the tests in a specific directory, you can give a list of specific directory names

```
pdk run /where/your/tests/are
```

- You can give the name(s) of specific files that contain your tests

```
pdk run xyz*.py
```

All of these cases create or append to a Pandokia log file, which contains the results from all the tests.

There is also a command "pdkrun" that you can use in place of "pdk run". It exists principally so that I can type:

```
!pdkrun
```

to repeat the last pdkrun command.

4.2 pdk run –recursive *directory*

When you give `--recursive`, pandokia recursively descends into the directory tree that you specify. In each directory, pandokia finds and runs tests by executing the command "pdk run *directory*". If no tests are found in a directory, zero test results are reported from that directory.

The configuration variable `exclude_dirs` contains a list of directory names to skip. (see pandokia/default_config.py)

You can specifiy multiple processes with the option `--parallel N` or the environment variable `PDK_PARALLEL`. It will execute tests in that many directories concurrently. It will not execute multiple concurrent processes in a single directory because we have found that the tests often interfere with each other. (This may be a characteristic of our test environment, which depends heavily on input and output files.)

4.3 pdk run *directory*

When you give the name of a directory, it compares the name of each regular file (not directories or device files) in that directory with a set of glob patterns that identify tests.

The file is disabled if the same base file name exists with the extension ".disable" or ".$PDK_CONTEXT.disable".

For example, if your test is test_xyz.py, it will be disabled if

- there is a file test_xyz.disable
- there is a file test_xyz.foo.disable and the currently running context is "foo"
- there is a file test_xyz.default.disable and the currently running context is "default"

If the Test-Runner knows how to report disabled tests, report will contains status=D (for disabled) for each test in that file.

Some Test-Runners do not know how to report the names of disabled tests. The *nose* and *py.test* Test-Runners included in the Pandokia distribution do.

If the file name looks like a test and it is not disabled, the test is executed by the same code that implements `pdk run *filename*`.

4.4 pdk run *filename*

When you explicitly give the name of a specific file, pdk run executes the tests in that file. It runs the tests even if the .disable file exists.

4.5 pdk run arguments and environment variables

`pdk run` can take parameters as environment variables and as command line arguments. Arguments always override the value in an environment variable.

Except as noted, all of the options can be used with any of the variations of `pdk run`.

--log or PDK_LOG

The series of test results will be written into this file, for subsequent import into the database. Default value is "PDK_DEFAULT.LOG."+test_run

–parallel or PDK_PARALLEL

Run up to this number of tests concurrently (but it will run at most one test at a time in any given directory). Only used with the -r (recursive) flag. Default value is 1.

--project or PDK_PROJECT

Use this as the project name. Default value is "default".

–test_run or PDK_TEST_RUN

Use this as the name of the test run. Default value is a generated string including the user name and the time to the nearest minute.

PDK_TIMEOUT

This environment variable sets the max number of seconds that a test runner can run. If set, individual test processes will be killed when they exceed this age.

This timeout applies to all the tests in a single file, not the individual tests. If you need timeouts for specific tests, you must use a test runner that implements per-test timeouts (such as py.test with the Pandokia plugin) or implement a timeout feature in your test code (possibly using a library such as *fixtures*).

Our normal use is to set PDK_TIMEOUT in a pdk_environment file. We have different timeouts in different directories.

All other environment variables with names beginning PDK_ are reserved for internal use by pandokia.

4.6 Monitoring the running tests

pdkrun can run multiple processes concurrently. To see a report of what is currently running, you can enter this command in the directory where you started the tests

```
pdk runstatus
```

This clears the screen and shows three columns of information

```
- the process "slot"
- date/time of last update to that process slot
- file name of tests executing in that process slot
```

The information is recorded in a file named *pdk_statusfile*.

If you set PDK_STATUSFILE to 'none', pdkrun will not record the status and the runstatus command will not work. (Later, this will be a way to say the name of the file to use.)

not implemented on Windows

4.7 Creating a Test Tree

Pandokia will preserve the hierarchy of your test tree as part of the test name. You can populate the directory tree with files containing tests in any organization that makes sense for your project.

The test running concurrency operates at directory granularity; so do the environment and contact files. You may wish to take this into account when creating the tree.

Place an empty file named pandokia_top at the top of the directory tree.

4.7.1 Overhead files

pandokia_top This is an empty file marking the top of the directory tree.

pdk_environment This is an INI-style file that may be used to customize the environment for the tests in this directory. It should contain named sections. The [default] section will apply to all tests; additional sections based on operating system ([os=foo] or [osver=foo]), machine architecture ([cpu=foo]), or hostname ([hostname=foo]) may also be included, and are applied hierarchically in that order.

Specifications of OS, version, or architecture are expected to be site-specific. We implemented a mapping that makes sense in our system; you may wish to examine and/or customize the env_platforms.py file.

The resulting environment will be merged with os.environ prior to running tests; in particular, any PATH environment variable is handled specially, and appended to (rather than overriding) existing values at a higher level.

pdk_contacts This file may be used to specify the username or email address of individuals (one per line) who should be notified about anomalous results for tests contained in this directory. The run command does not read this file; see database.rst, "Importing Contacts" for more detail.

4.7.2 Writing a nose test

TODO: move this into the runners_nose section

TODO: refer to a directory of sample tests

Pandokia will support any type of test that nose supports: unittests, doctests, and arbitrary test functions that raise assertion errors if they fail.

unittest/testcase style

This example shows how to add attributes to a unittest-style test.:

```
class BasicTest(unittest.TestCase):
    def setUp(self):
        self.tda={}
        self.tra={}
        self.tda['foo']='bar'

    def test1(self):
        self.tda['func']='add'
        self.tra['sum']=4+2

        # If the assertion fails, the test fails.
        self.assert_(4+2==6)
```

Any old function

Test functions can be written as follows:

```
#TDAs and TRAs are supported via global variables. The
#plugin takes care of clearing them so there is no crosstalk
#between tests.

tda = dict()
tra = dict()

def testxyz() :
     tda['cat']='tortoiseshell'
     # If the assertion fails, the test fails
     assert True

def testabc():
     tda['func']='add'
     sum=4+2
     tra['sum']=sum
     assert sum == 6
```

(continues on next page)

(continued from previous page)

```
def testglobal():
     global tda
     tda = {'cat':'lion'}
     #The global statement is necessary in order to avoid rebinding
     #rebinds the name to a local variable, which will not be seen
     #by Pandokia
     assert True
```

doctest

#TDAs and TRAs are not supported in doctests. “”” >>> print 1+1 2

```
>>> print 7-3
4
"""
```

4.7.3 Using file comparators

TBD, but see `example_filetest.py`.

CHAPTER 5

Pandokia Test Result File Format

TODO: move this to Adding Test Runners section

A test report file is a series of one ore more test result records, with each record represented by several lines of ASCII text that comply with the following format:

```
name=value
    indicates a one-line value for <<name>>

name:
.line
.line
.line
    \n
    indicates a multi-line value for <<name>>.  There
    is a newline immediately after the colon, and each
    following line begins with a period.  A blank line
    ends the field.  The \n after the ':' is not part
    of the string.  The blank line at the end is not
    part of the string.  The '.' at the beginning of
    each line is not. Because of database limitations,
    nul characters are converted to \0 when imported into
    the database.

Blank lines and comment lines (beginning with "#") are ignored.
```

The following special commands are also recognized:

START The START command will reset the state of the input processor. This includes clearning any defaults. You can write "nnSTARTn" to ensure a clean state when appending to a file.

END REQUIRED. Indicates end of a record; the record is entered in the database. This command must be present for the record to be imported.

SETDEFAULT this record is not entered in the database, but the values are filled in as default values for all following records. You can still set a field, even if there is a default. The last value you set will be used.

TODO:

CHAPTER 6

What are the contents of a test result?

There is one test_result each time we run a specific test. The fields can be categorized as follows

Type: This set of fields identifies a type of test:: project test_runner test_name

Identity: this set of fields identifies a single test_result record:: test_run host context project test_name

Required: This set of fields is required:: test_run project test_name status

Optional: All other recognized fields:: location (of the test file) test_runner (used to run the test) log (stdout/stderr logs) start_time end_time tda_* tra_*

An unrecognized field will not cause an error, but will not be imported into the database.

CHAPTER 7

What are the meanings of the fields in a report record?

(test_run, project, host, context, test_name) identify the specific instance of a test result.

- test_run identifies a specific instance of running the test. If you run a test suite more than once, you need some way to distinguish which run a result came from. Each time you run it, you report it as a different test_run. Normally, you would find the same test name with many different host/context values, indicating that you ran the same test in many environments.

  The reporting system displays the name of the test_run that a result came from. It can also compare the results of multiple test runs. e.g. This test failed today, but it passed yesterday.

- project is an arbitrary way of grouping tests. The authors have many separate projects that are tested each night as part of our continuous integration. In the morning, we look at a single report that provides a summary of the results from all the projects.

- host identifies the machine where this test ran.

- context identifies some specific configuration. For example, if we run the same test in two different versions of python, we use a different context to report each. The context can identify versions of interpreters or shared libraries, cpu type, available memory – it means whatever you want it to mean. You say the name of the context when you run the test, and you read the name of the context in the reports.

- test_name identifies the test. A test name is unique within a project. The test name is the same every day and in every execution environment, so that you can compare results from day to day or across environments.

These fields uniquely identify a test result. It is an error to report the same result again, but if you run the test again, the test is part of a different test_run so you have a different test result.

Other than the identity, there is only one required field:

- **status tells us what happened with the test**

  P = passed the test observed whatever it expected

  F = fail we ran the test, but it failed

  E = error For some reason, we could not complete the test. We distinguish this from Fail because an error may indicate a problem with the test, not a problem with the software being tested.

  D = disabled We asked not to run the test. We report this so that the test is not Missing.

M = missing We did not receive a report, but expected one. You don't normally report Missing in a test result file, but this condition can be detected later in the database.

There are various optional fields:

- test_runner tells us which software ran the test. Since we can accept reports from many test systems, this tells us which one produced this result.
- start_time
- end_time The cumulative run time of some of our test suites is many hours. It helps to know when a particular test ran.

  Two formats are supported for these timestamps:

```
- time_t (seconds since 1970)
  may be floating point for fractions of a second

- YYYY-MM-DD HH:MM:SS.sss
  The output from
     date '+%Y-%m-%d %H:%M:%S'
  matches this format.  All times in this format are assumed
  to be local time.
```

- location is information for a developer who has to diagnose a problem with a test. Since we have 5 projects spread over 8 test environments, and a total of nearly 5000 tests, it is helpful for the report to display the location of the one you are looking at. The location is normally the fully qualified file name of the test.
- log is the stdout/stderr from a test. If the test fails, it helps to see what it said.
- tda_* tra_*

  TDA/TRA (test definition/result attributes) are further information that the test explicitly reports. The test author can make up names for as many TDA or TRA fields as they want. The names will be converted to lower case on import to the database.

  The meaning of a TDA or TRA field is only defined for a specific test, though as test author you are free to use the same meanings for a group of tests. It is information for the developer to analyze what was happening. An attribute may be useful even for a test that passed, if it tells you something about how/why/how-well it passed.

CHAPTER 8

Test Runners

abstract This is a summary of the test runners included with pandokia.

8.1 any - shell_runner - shell scripts or simple programs

abstract `shell_runner` executes a shell script or simple program that contains a single test. The exit status of the script is the test result. For `sh`, there are helper functions to compare files and report test attributes.

Contents

8.1.1 general

With `shell_runner`, each test file contains a single test. The name of the test is the name the test file with the extension removed.

The exit status of the script is the status of the test:

- exit code 0 is Pass
- exit code 1-127 is Fail

- exit code 128-255 is Error (this range of exit codes is normally associated with a process that exits due to an un-trapped signal.)

The entire stdout and stderr of the script are captured and reported in the test result.

There are examples in the source code in pandokia/sample_tests/shell

8.1.2 sh

Special helper functions are available in sh, bash, and compatible shells:

```
. pdk_shell_runner_helper
```

Your script can create an output file and compare it to a reference file.:

```
init_okfile

testfile cmp  $file
testfile diff $file

exit $teststatus
```

Your script can report attributes:

```
bug: not implemented
```

8.1.3 csh

In general, sh is a better scripting language than csh, but you can write your test in csh if necessary.

There are examples in the source code in pandokia/sample_tests/shell

8.1.4 other shells

Files that are executable are executed directly with `./filename` . If you want something other than sh or csh, you can use #! to call out a specific shell.

8.1.5 reporting attributes

In `sh`, there are helper functions described above that log test attributes.

The environment variable PDK_LOG contains the name of the pandokia log file. Any test can append tda/tra values directly to the log file.

8.2 any - maker - compile a test and then run it

Contents

8.2.1 tbd

8.3 any - run - run an external program, for testing compiled code

abstract Test frameworks for compiled languages usually require you to compile a special test program. To use one of these, you can have your build system (make, Ant, whatever) compile and install some number of test programs, then use this pandokia runner to execute them.

Your test framework must write Pandokia-formatted results to the PDK_LOG file. (If you have a program that returns an exit code for pass/fail, see the shell_runner runner.)

Contents

8.3.1 Running plain programs

The "run" test runner just executes a program. It assumes that the program knows how to make a pandokia report. You can use this for special cases and for installed test programs that were compiled as part of some external build process.

Compiled Separately

The build process for your system under test can compile and install test programs that are pandokia aware. You can then use the "run" runner to execute those external programs.

It may be helpful to place a shell script in your test directory:

```
#!/bin/sh
exec my_test_program
```

where my_test_program could be somewhere on PATH.

Special Shell Scripts

If you want to implement tests procedureally instead of through a test framework (such as shell_runner or shunit2), you can write a shell script that performs the test.

For shell scripts, there is a library named "pdk_run_helper.sh". The basic outline is:

```
. pdk_run_helper.sh

# begin a test named "P".
test_start P
    # do some stuff here
    echo this is test P
    # report a test status - you can do this as many times as
    # you like; the resulting status will be the worst case that
```

(continues on next page)

(continued from previous page)

```
    # was reported
    test_status P
# end of test
test_end

test_start this_one_fails
    # how to set an attribute
    test_attr tda_this yes
    test_attr tra_that no
    # determine test status
    if false
    then
        test_status P
    else
        test_status F
    fi
test_end

# call this at the end to remove temp files
cleanup
```

8.4 C - fctx - unit tests in C and languages callable from C

8.4.1 Overview

FCTX is a C unit test framework. Everything you need to compile and run tests is included with Pandokia. It used to come from http://fctx.wildbearsoftware.com/ , but that site has been down for some time, so we have documentation at http://ssb.stsci.edu/testing/fctx . The github repository is at https://github.com/imb/fctx .

In FCTX terms, the Pandokia interface is implemented as a "custom logger". To make your tests Pandokia-capable, you need to include pandokia_fct.h instead of fct.h, which causes the custom logger to be installed.

Here is a simple example:

```
#include "pandokia_fct.h"

FCT_BGN()
{
    FCT_QTEST_BGN(test_name_1)
    {
        printf("This test will pass\n");
        fct_chk(1);    // pass
    }
    FCT_QTEST_END();

    FCT_QTEST_BGN(test_name_2)
    {
        printf("This test will fail\n");
        fct_chk(0);    // fail
    }
     FCT_QTEST_END();

}
FCT_END()
```

This example is minimal, though it is adequate for many purposes. In principle, you can also use the more advanced features of FCTX such as fixtures, suites, conditional tests, and advanced checks, though those features are not heavily tested with Pandokia. See the FCTX documentation for details.

8.4.2 Compiling the test program

The files fct.h (the fctx test framework) and pandokia_fct.h (the pandokia logger) are included in the pandokia distribution. The command

```
pdk maker
```

will say the name of the directory where you can find the files. So, for example, if you paste the example code above into mytest.c, you can compile it with:

```
cc -o mytest -I`pdk maker` mytest.c
```

8.4.3 Running the test program outside Pandokia

A compiled test program is a normal FCTX test. You can just run it to see output in the FCTX format.

```
./mytest
```

If the test program sees the environment variable PDK_FILE, the Pandokia logger will assume it is being run by pandokia and take over the logging function.

You can explicitly ask it to make pandokia output by specifying "–logger pdk".

```
./mytest --logger pdk
```

The pandokia plugin does not accept any command line parameters; it takes values from the environment, or it uses defaults. The default pandokia log file is named "PDK_LOG".

8.4.4 Using with the "run" test runner, option A

The "run" test runner just runs an external program that is pdk-aware. You would have your normal build system compile/install the test somewhere and then define the test as a shell script something like this:

pdk_runners:

```
*.fctx    run
```

xyzzy.fctx:

```
#!/bin/sh
# This file must be here and executable for pandokia to know about
# the test, but it just runs the actual test program.
/install/dir/mytest
```

The names of the tests will be prefixed with the base name of the file that pandokia finds, not the name of the executable that contains the tests. In this example, there are two tests found: xyzzy/test_name_1 and xyzzy/test_name_2.

8.4.5 Using with the "run" test runner, option B

You could have your build system compile the tests directly in the directory of tests, so that you might run the commands

```
make all
pdkrun .
```

To do this, list each file in pdk_runners:

```
test_one    run
test_two    run
```

and use a makefile like this:

```
all: test_one test_two

test_one: test_one.c
    cc -o test_one -I `pdk maker` test_one.c

test_two: test_two.c
    cc -o test_two -I `pdk maker` test_two.c
```

8.4.6 Using with the "maker" test runner

If you use the "maker" test runner, you may be able to include your C source code directly in the test directory. Pandokia can compile and run it for you.

pdk_runners:

```
*.c    maker
```

mytest.c:

```
/*
* $ cc -o mytest -I`pdk maker` mytest.c
* $ ./mytest
*/
... rest of the C code for your test ...
```

8.4.7 Capture of stdout/stderr

This test runner will capture stdout/stderr of your tests, but the underlying fctx implementation has a bug. It directs stdout/stderr into a pipe, then reads the pipe back *in the same process* to collect the output for logging. If your test prints more output that fits in a pipe, the test will deadlock writing to the pipe.

8.5 sh - shunit2 - an xUnit test framework for bourne shell scripts

8.5.1 Overview

This runner uses a modified version of shunit2; the modifications implement support for a plugin architecture.

The plugin that reports to the Pandokia system is distributed with Pandokia.

(As of this writing, patches to implement this plugin architecture have been submitted to the shunit2 maintainer, but have not been accepted into the source code.)

8.5.2 Installing

You can download the patched shunit2 from http://ssb.stsci.edu/testing/shunit2

Download the file "shunit2" and put it on your PATH somewhere. No changes to your installed Pandokia are required.

This copy of shunit2 is identified by SHUNIT_VERSION='2.1.6plugin'

The *unmodified* documentation for shunit2 is available at https://shunit2.googlecode.com/svn/trunk/source/2.1/doc/shunit2.html or http://ssb.stsci.edu/testing/shunit2/shunit2.html

8.5.3 Using shunit2 with Pandokia

This version of shunit2 can use the command "shunit2 xyz.shunit2" to run the tests in a file. Pandokia uses this form exclusively.

The file of tests should NOT use shunit2 as a library. This has the odd implication that the tests distributed with shunit2 do not run unmodified in pdkrun.

With earlier versions of shunit2, it was conventional to write tests as a program that include shunit2 as a library. This mode is not tested with Pandokia.

Basic Example

For use with Pandokia, write a shell script that has function names starting with "test". For example,

```
test_one_equals_one()
{
    echo 'this one passes'
    assertEquals 1 1
}

test_two_equals_one()
{
    echo 'this one fails'
    assertEquals 2 1
}
```

Save this test with a file name that ends xyz.shunit2 and run pandokia on it:

```
pdkrun xyz.shunit2
```

Assertions in shunit2

The shunit2 distribution contains documentation of various assertions in doc/shunit2.rst ; you can read it online at http://shunit2.googlecode.com/svn/trunk/source/2.1/doc/shunit2.html or http://ssb.stsci.edu/testing/shunit2/shunit2.html

There is significant difference of interest to python programmers: When an assertion fails in shunit, it does NOT abort the rest of the test function. It sets a failure flag and continues execution. This means that you can have arbitrarily many assertions in a single test function. The result of that test will be Fail if *any* of the assertions fail.

```
test_foo()
{
    # this test fails because of the second assertion
    assertEqual 1 1
    assertEqual 2 1
    assertEqual 3 3
    echo 'test_foo finished'
}
```

You can also explicitly declare a test to fail with the "fail" function:

```
test_bar()
{
    fail "this test always fails"
}
```

Erroring Tests

shunit2 normally considers a test to have a status of Pass or Fail; it does not natively have the concept of Error.

If you want to report a status of Error to the Pandokia system, you can call the special function pdk_error:

```
test_baz()
{
    pdk_error "this test errors"
}
```

Disabling individual tests

```
test_foo() {
    echo 'in shunit, you can produce output from a skipped test'
    _shunit_assertSkip
    return
    echo pass
}
```

Pandokia Attributes

You can report tda/tra attributes with the pdk_tda or pdk_tra functions:

```
test_with_attr()
{
    pdk_tda one 1
    x=`ls | wc`
    pdk_tra filecount $x
    pdk_tra foo
}
```

Using pdk_shell_runner_helper

If you do not have reference files:

```
. pdk_shell_runner_helper

test_name1() {
    # must init the helper at start of each test
    init

    # declare any tda attributes
    pdk_tda foo 1

    # do something
    thing=`echo X`

    # report a test result
    case "$thing"
    in
    pass)
            :       # do nothing special to indicate pass
            ;;
    fail)
            fail    # regular shunit2 way of failing a test
            ;;
    *)
            pdk_error # how to declare error to shunit2
            ;;
    esac

    # declare any tra attributes
    pdk_tra bar 2
}
```

If you have reference files to compare:

```
. pdk_shell_runner_helper

test_name2() {
    # You must init the helper at start of each test; this does all
    # the regular init AND declares the okfile for tracking
    # output/reference files.

    init_okfile ${_shunit_test_}

    # Make some output files.

    echo hello > out/${_shunit_test_}.f1
    echo world > out/${_shunit_test_}.f2

    # Use testfile to compare the output to the reference file.
    # testfile declares the pass/fail/error status to shunit2
    # and pandokia.

    testfile diff out/${_shunit_test_}.f1
    testfile cmp  out/${_shunit_test_}.f2

    # you can declare attributes
    pdk_tda foo 1
    pdk_tra bar 2
}
```

shunit2 outside pandokia

To make your shunit2 tests work in or out of pandokia:

```
. pdk_shell_runner_helper

test_whatever() {
    ...
}

if [ "$SHUNIT_VERSION" = "" ]
then
    . shunit2
fi
```

If you write your tests in this form, you can run them with any of these commands:

```
pdkrun foo.shunit2

shunit2 foo.shunit2

./foo.shunit2
```

installed shunit2 tests

You can write shunit2 tests that are installed on the users PATH. The user can then run them by typing the name, but it requires special handling to have pdkrun find and execute them.

Use the "run" runner. Create file.run containing:

```
#!/bin/sh
exec shunit2 --plugin pdk installed_name.shunit2
```

shunit2 extended capabilities

This modified shunit2 contains some new features.

You can list all the test names that are defined in a shunit2 file:

```
shunit2 file.shunit2 --list
shunit2 file.shunit2 -l
```

You can specify a list of tests to run, in place of all the tests in the file:

```
shunit2 file.shunit test_1 test_2
```

8.6 SPP - regtest - an STScI legacy system for IRAF packages

8.6.1 Overview

`regtest` tests IRAF tasks. It is derived from a legacy system, and is still in use at STScI. It is so specialized that it is unlikely to be useful to you.

8.6.2 Test definitions

Each test exists in a separate XML file. The file format is defined in the Pandokia source code in stsci_regtest/TEST_FORMAT.xml

regtest automatically generates the data needed to make okifiable tests.

8.6.3 Test Execution

This runner uses PyRAF to execute IRAF tasks. Before executing the tasks under test, it loads these IRAF packages:

```
fitsio
images
stsdas
tables
```

In the test definition, it lists one or more output files. Each output file is compared to a reference file. If all match (within the parameters of the comparison), the test passes.

8.7 Python - minipyt - a simple but reliable test runner, built in to Pandokia

abstract minipyt (mini python test) runs tests written in python. The tests can be either functions, classes, or simple code sequences managed by the "with" statement. For test classes, you can have one instance for each test method or you can have a single instance shared by all test methods.

A major advantage of minipyt is that any test file __always__ reports a test result of some kind, no matter what kinds of errors occur in that file. Even a python file that fails to import because of syntax errors will report at least one record. This differs from nose and py.test because the plugin architecture of those systems is not able to report certain kinds of errors.

Contents

        * *prevent using nose by mistake*
    - *Decorators*
        * *minipyt*
        * *nose*

8.7.1 Importing the Test File

minipyt runs tests by importing a python file, searching for tests, then running those tests.

There is a test record for the act of importing the file. For example, xyz.py will have a test name of "xyz". The status value reported is:

- Pass means that minipyt was able to import the file and examine it for tests. If the file executed any code during the import, it succeeded
- Error means that an exception was raised while importing the file or examining it for tests.
- Fail means that AssertionError was raised during the import.

For pure python, minipyt will always report at least one test record for each file, no matter how badly the import goes. (If the test uses a C extension that core dumps the python interpreter, there is still a possibility to lose the test report.)

It is possible to consider a python file to be a single test. The simplest minipyt test that passes is:

```
assert True
```

and the simplest test that fails is:

```
assert False
```

In these examples, the name of the test matches the name of the file.

You can report attributes for the module by loading them into the global dictionaries module_tda and module_tra.

8.7.2 Recognizing Tests

After the import (which may execute a test directly), these things in the namespace of the imported module are recognized as tests:

- functions that have names beginning or ending with "test", that are not decorated with `pandokia.helpers.minipyt.nottest` or `nose.tools.nottest`
- classes that have names beginning or ending with "test" that are not decorated with `pandokia.helpers.minipyt.nottest` or `nose.tools.nottest`
- functions decorated with `pandokia.helpers.minipyt.test` or `nose.tools.istest`
- classes decorated with `pandokia.helpers.minipyt.test` or `nose.tools.istest`

By default, tests are executed in order by line number. If you want to order the tests by name, set the global variable:

```
minipyt_test_order = 'name'
```

in your test module.

Sorting by line number is unreliable for decorated functions because the line number used for the sort may be the line number of some part of the decorator, not the line number of your code.

8.7.3 Executing Test Functions

Test functions are executed by calling the function with no parameters. The test passes if it returns, fails if it raises AssertionError, and errors if it raises any other exception.

If the function is decorated with a setup function, the setup is called before the test function. Any exception from the setup is handled the same as if it were raised by the function itself.

If the function is decorated with a teardown function, the teardown is called after the function. This happens even if the test function terminated with an exception. Any exception in the teardown function will cause the test to report the status Error.

8.7.4 Executing Test Classes

A Test Class defines an object with tests defined as class methods. A method is a test if the name starts or ends with "test" and has not been decorated with nottest, or if it has been decorated with pandokia.helpers.minipyt.test or nose.tools.istest.

single object instance

If the class has the attribute minipyt_shared=True, only one object instance is created; all the tests run in that same instance. This can be helpful if the class initialization performs some shareable operation, such as connecting to a database.

```
- create object

- call obj.classSetUp()

- for each test method

    call obj.setUp()
    call test method
    call obj.tearDown()

- call obj.classTearDown()
```

obj.class_tda and obj.class_tra are attributes that belong to the class.

obj.tda and obj.tra are attributes that belong the the most recently executed tests. The attribute set reported is the union of class_tXa and tXa.

multiple object instances

If the class has the attribute minipyt_shared=False or if the attribute is not set, a new instance is created for each test. This is similar to other test frameworks such as nose, py.test, and unittest.

```
- for each test method

    create obj

    call obj.classSetUp()

    call obj.setUp()
```

(continues on next page)

(continued from previous page)

```
    call test method

    call obj.tearDown()

    call obj.classTearDown()
```

obj.class_tda and obj.class_tra are attributes that belong to the class.

obj.tda and obj.tra are attributes that belong the the most recently executed tests. The attribute set reported is the union of class_tXa and tXa.

running the test methods

minipyt will call each method to execute the test. If the class has a method named "setUp", it will be called before each test method. Any exceptions or assertions in the setUp method will have the same effect as if they happened in the test function.

If the class has a method named "tearDown", it will be called after the test method. This happens even if the test method terminated with an exception. Any exception in the teardown method will cause the test to report the status Error.

The names setUp and tearDown are compatible with nose. The nose.tools.with_setup decorator has no effect on class methods.

8.7.5 Linear Execution in Sequential Code ("with" statement)

When python imports a file, the code in that file is executed. minipyt can report that as a single test, or you can use "with" statements to divide it into multiple tests.

This file contains two tests named "foo" and "bar":

```
import pandokia.helpers.pycode as pycode

with pycode.test( 'foo' ) as t:
    pass

with pycode.test( 'bar' ) as t:
    assert False
```

You can set attributes on the test by assigning to the tda or tra dicts in the context manager:

```
with pycode.test( 'baz' ) as t:
    t.tda['yes'] = 1
    t.tra['no']  = 0
    assert 2 + 2 = 4
```

Tests that are defined in "with" statements can be nested:

```
with pycode.test( 'foo' ) :
    # this test is named "foo"
    print "set up in foo"
    with pycode.test( 'bar' ) :
        # this test is named "foo.bar"
        print "bar"
        assert 2 + 2 = 5
```

(continues on next page)

(continued from previous page)

```
        print "if we got here, not in room 101"
    print "more output in test foo"
```

Tests that are defined in "with" statements may be used inside test functions:

```
def test_plover() :
    # this test is named "plover"
    print "just a plover"
    with pycode.test( 'egg' ) :
    # this test is named "plover.egg"
        print "An emerald the size of a plover's egg"
        with pycode.test( 'hatch' ) :
            # this test is named "plover.egg.hatch"
            assert 1
    print "that laid an egg"
```

You can use this feature to dynamically define tests:

```
for x, y  in some_list :
    with pycode.test( str(x) ) :
        assert f(x,y)
```

This example is similar to the parameterized tests in py.test, but you do not need to have the entire list of tests before the tests start running.

why not py.test parameterized test or nose generators?

If those methods are more convenient, you should use them. Here are some features that can be an advantage of this approach:

- the simplicity of linearly executing procedural code: There are no callbacks, no implicit ordering, no separate setup/teardown functions to keep track of.
- you choose the test name; in a parameterized test or a generator, all the parameters to the test function are included in the test name, even when they are not all relevant. The pandokia plugins for pytest/nose cannot know which parameter values may be excluded from the name, so they include them all.
- easy setup/teardown

```
with test('group') :
    db = sqlite3.connect('test.db')
    with test('first') :
        ...
    with test('second') :
        ...
    db.close()
```

- arbitrarily deep nesting: By nesting "with test()" statements, you can build aribitrarily deep test hierarchies, if it is suitable for your application.

8.7.6 Special Features

dots

minipyt is normally silent when it runs tests. If you want it to print dots, you can

- set the environment variable PDK_DOTS
- set the module variable minipyt_dots_mode

to one of these values:

- a zero-length string gives the default behaviour
- 'S' shows a dot for each passing test and the status for any non-passing test
- 'N' shows a dot for each passing test and the test name and status for any non-passing test
- 'O show a dot for each passing test and the test name, status, and output for any non-passing test

If you specify both the environment variable and the module variable, the module variable takes precedence.:

```
# no dots
minipyt.dots_mode = None

# show dots and the name+status of the non-passing test
minipyt.dots_mode = 'N'
```

prevent using nose by mistake

nose should recognize and execute many minipyt tests, but you can explicitly prevent using a test file with nose by:

```
import pandokia.helpers.minipyt as mph
mph.noseguard()
```

`noseguard()` raises an exception if 'nose' is in sys.modules.

This prevents importing the file if nose is also loaded. If pandokia is using minipyt as the test runner, nose will not have been imported. If nose is in sys.modules, we assume that is because the test file was mistakenly run using nose.

Presumably, this may cause you problems if you are trying to import the test into an interactive python. If so, disable this function with:

```
import pandokia.helpers.minipyt as mph
mph.disable_noseguard = True
```

8.7.7 Decorators

minipyt

These decorators are available in pandokia.helpers.minipyt:

- `test`

  marks a function or class as a test, even if the name does not otherwise look like a test

- `nottest`

  marks a function or class as not a test, even if the name looks like it should be a test

All work on both functions, classes, and methods.

nose

Many nose decorators work in minipyt tests.:

```
import nose.tools

@nose.tools.raises(IOError)
def test_mine() :
    ...
```

These decorators are known to work:

- `nose.tools.raises`
- `nose.tools.timed`
- `nose.tools.with_setup` (on test functions only, not class methods)
- `nose.tools.nottest`
- `nose.tools.istest`

8.8 Python - nose - run tests with nose

abstract Pandokia can use nose 0.11.1 or 1.0 to run tests. It uses a nose plugin to gather the test results. There are some limitations.

Contents

8.8.1 Overview

TODO: Write this section.

Really, you just write nose tests in a python file, then use "pdkrun file.py" or "pdkrun -r ." or something to run them. Each file of tests runs is a different process. There is no good way to pass advanced nose options through pandokia, but we typically find that we don't need to. If you want to debug a test, you can do that outside pandokia with something like "nosetests –pdb file.py"

8.8.2 Disabling individual tests in nose

It is hard to find this in the nose documentation:

```
from nose.exc import SkipTest

def test_foo() :
    raise SkipTest('busted')
```

8.9 Python - pytest - run tests with py.test

Contents

8.9.1 Overview

Pandokia supplies a py.test plugin. It is automatically enabled when you use Pandokia to run a test with py.test.

With this plugin, py.test can also write a Pandokia log file on its own, so you can run it outside Pandokia.:

```
py.test --pdk foo.py
```

8.9.2 Installing

Documentation for installing py.test is available at http://pytest.org/ . The py.test documentation includes quite a nice Installation and Getting Started section. If you are not already familiar with py.test, it is a good introduction.

8.9.3 Enabling py.test

Any file matching the pattern **.pytest* will be executed with py.test.

The default runner for **.py* is nose, but you can override this per-directory by creating a *pdk_runners* file.

The most convenient way to do it is to declare that all python files should be run with py.test:

```
*.py        pytest
```

You can also mix py.test, nose, and other test runners:

```
a.py        pytest
b*.py       nose
c*.py       minipyt
d*.py       pytest
```

8.9.4 Test timeouts

You can define timeouts for individual tests like this:

```
@pytest.mark.timeout(3)
def test_foo() :
    # this test always times out
    time.sleep(6)
```

The timeout is always specified in seconds. If the test runs longer than that, the plugin will trigger SIGALRM that will raise an exception.

The plugin attempts to detect the application program setting a signal handler for SIGALRM. If a signal handler already exists for SIGALRM when the test starts, or if the test ends with a signal handler that is not the one that the plugin installed, the test will report an error. The exception will identify the signal handling function.

Timeouts are not implemented on Windows.

If the Pandokia plugin is not enabled, the timeout markup has no effect.

This feature is independent of PDK_TIMEOUT.

8.9.5 Knowing your test name

You can tell your test function the name that Pandokia knows it by like this:

```
def test_foo(pdk_test_name) :
    print "My pandokia test name is: ", pdk_test_name
```

You could use this in data that is reported outside the Pandokia system. For example, when we test a web app, we sometimes include the test name in the browser string so that we can later identify the actions of specific tests in the httpd server logs.

If the pandokia plugin is present but not enabled, the value of pdk_test_name is None.

If the pandokia plugin is not present, pdk_test_name is an unrecognized funcarg and therefore it is an error to use it.

8.9.6 Capture of stdout/stderr

The plugin captures stdout/stderr by replacing sys.stdout and sys.stderr with StringIO.

More work is needed on this feature. Currently, it disables the capsys/capfd capability that is normally present in py.test.

If the pandokia plugin is not enabled, capsys/capfd work as normal for py.test.

8.9.7 Disabling individual tests in py.test

```
@pytest.mark.skipif("True")
def test_me() :
    pass
```

Bug: As of pandokia 1.2rc6, this causes the test to be reported as “Pass” instead of “Disabled”.

8.10 Python - unit2 - unittest2

abstract Runs tests with the Python unittest2 0.5.1 package.

Contents

8.10.1 overview

This runner was written to try interfacing with unittest2. It seems to work, but in the end we did not find unittest2 attractive enough to be worth using. We have py.test and nose.

You can find unittest2 at http://pypi.python.org/pypi/unittest2

CHAPTER 9

Useful Patterns and Helper Functions

v. index:: single: patterns; xml; alternative for new tests .. index:: single: patterns; regtest; new

9.1 Python: Tests based on reference files

abstract One way to write tests is to create some output files and then compare them to reference files. This is often a form of regression test, where the reference is output from a previous run.

This section describes a way to implement these tests in python using tools in pandokia.helpers

9.1.1 Basic outline

You write a test based on reference files in the same way as any other python test, except that your assertions are based on the result of file comparisons.

Here is a basic template that works in py.test, nose, and minipyt:

```
import pandokia.helpers.filecomp as filecomp

def test_foo():

    # define your test name here
    testname = 'test_foo'

    # put this exact code here
    global tda, tra
    tda = { }
    tra = { }

    # Define the list of output/reference files
    output = [
        ... details described below
```

(continues on next page)

(continued from previous page)

```
        ]

    # Delete all the output files before starting the test.
    filecomp.delete_output_files( output )

    # Run the code under test here
    # CALCULATE SOME THINGS OR CALL SOME FUNCTIONS

    # If the code under test isn't doing this already, write the results
    # to one ore more output files (defined above) for comparison
    f = open("out/file","w")
    f.write("something\n")
    f.close()               # be sure to close the output file

    # compare the output files - use this exact command
    filecomp.compare_files( output, ( __file__, testname ), tda = tda, tra = tra )
```

Here is a detailed commentary:

```
def test_foo():

    # define your test name here
    testname = 'test_foo'
```

The test name is used in the call to compare_files; it is used to create a unique file name that is used by the FlagOK feature.

```
# put this exact code here
global tda, tra
tda = { }
tra = { }
```

The tda is needed to report the location of the okfile.

```
# definition of output/reference files
output = [
    ... details described below
    ]
```

You can also add your own attributes if you want. If the code under test takes input parameters that make this test special, those input parameters are a good choice.

You need a list of output files, reference files, and how to peform the comparison. This is somewhat involved, so it is described below.

```
# Delete all the output files before starting the test.
filecomp.delete_output_files( output )
```

Before starting the test, delete all the output files. If the code under test fails to produce an output file, it will be detected. (If you don't delete the output files, your comparison may see an old file that was left behind from a previous test run.)

```
# Run the code under test here
# CALCULATE SOME THINGS OR CALL SOME FUNCTIONS

# If the code under test isn't doing this already, write the results
```

(continues on next page)

(continued from previous page)

```
# to one ore more output files (defined above) for comparison
f = open("out/file","w")
f.write("something\n")
f.close()               # be sure to close the output file
```

Here you exercise the code under test to produce output files. The comparison of these files to the previously existing reference files is your test.

```
# compare the output files - use this exact command
filecomp.compare_files( output, ( __file__, testname ), tda = tda,)
```

This tool compares all the files. It raises an Exception if there is an error or AssertionError if one of the files does not match.

Defining the list of output files - simple form

To use the simple form, you must create your output files in the directory out/, which will be created for you.

Each element of the list is a tuple of (filename, comparator, comparator_args).

Create the file in the directory out/, but list only the base name of the output file here.

For example, if your test creates "out/xyzzy.fits", you can compare it to "ref/xyzzy.fits" using fitsdiff:

```
output = [
    ( 'xyzzy.fits', 'fits' ),
]
```

It is often useful to pass additional parameters to fitsdiff. List them in a dict in the third element of the tuple:

```
output = [
    ( 'xyzzy.fits', 'fits', { 'ignorekeys' : [ 'filename', 'date' ] } ),
]
```

If you follow the framework in this chapter, the out/ and ref/ directories will be created for you. You will need to create your own reference files, either by copying files into the ref/ directory or using the FlagOK feature in the GUI.

You can mix the two styles in a single list:

```
output = [
    ( 'xyzzy.fits', 'fits', { 'ignorekeys' : [ 'filename', 'date' ] } ),
    ( 'plugh.fits', 'fits' ),
    ( 'plover.fits', 'fits' ),
]
```

Defining the list of output files - complex form

The simple form requires a certain directory structure for your output and reference files. If you cannot adhere to that requirement, you can give a more detailed definition:

```
output = [
        {
            'file'      : 'A.fits',         # the name of the output file
            'reference' : 'A_ref.fits',     # the name of the reference file
            'comparator': 'image',          # the comparator to use
```

(continues on next page)

(continued from previous page)

```
            # additional args to the comparator
            'args'      : {
                'ignorekeys': [ 'filename', 'date', 'iraf-tlm' ],
             },

        },
]
```

9.1.2 Available Comparators

binary

This comparator checks that the files contain identical byte streams. It takes no additional args.

```
output = [
    ( 'xyzzy', 'binary' ),
]
```

diff

This comparator uses difflib to make a unified diff of two text files. This comparator reads both entire files into memory.

```
output = [
    ( 'xyzzy.txt', 'diff' ),
]
```

There is one optional parameter:

- rstrip (True/False)

  removes trailing white space from each line before the comparison

rstrip is useful if you might use json.dump() or pprint.pprint() to write out a more complex python data structure to your file. In some cases, json will write trailing spaces that are not significant.

```
output = [
    ( 'xyzzy.txt', 'diff', { 'rstrip' : True } ),
]
```

fits

This runs fitsdiff to compare the files.

```
output = [
    ( 'xyzzy.fits', 'fits', { 'maxdiff' : 1e-5 } ),
]
```

Additional arguments are :

- maxdiff (float)

  This is the fitsdiff maxdiff number specified by `fitsdiff -d`

- ignorekeys (list)

  This is a list of header keywords that are ignored. They are passed to `fitsdiff -k`.

- ignorecomm (list)

  This is a list of header keywords whose comments are ignored. They are passed to `fitsdiff -c`.

These additional arguments are the same as used in the stsci XML regtest system, but the lists are specified as python lists like ['a', 'b'] instead of as a single string like 'a,b'

text

This is the text comparison from the stsci XML regtest system. It does not make especially interesting diffs, but is has facilities to ignore various patterns in the text.

```
output = [
    ( 'xyzzy.txt', 'text', { 'ignore_wstart' : [ 'plugh', 'plover' ]  } ),
]
```

Additional arguments are :

- ignore_wstart (list)

  words that start with this text are ignored

- ignore_wend (list)

  words that start with this text are ignored

- ignore_regexp (list)

  this regex is ignored

- ignore_date (True/False)

  patterns that look like a date/time stamp are ignored; the system contains a rather elaborate regex to recognize many date formats

All this ignoring is performed by translating regular expression matches to the value " IGNORE ".

user-defined comparators

You can provide your own comparison function before you call filecomp.compare_files().

```
filecomp.file_comparators['mycmp'] = my_function

def test_1() :
    ...
    filecomp.compare_files( ... )
```

See the docstring for pandokia.helpers.filecomp.cmp_example for a definition of the interface to your comparator function.

9.2 Python: Replacing XML regtests with Python

It is fairly easy to translate an XML stsci regest into python. Here is an example.

An XML regtest:

```
<!--
##
## HEDIT - images/imutil/hedit: level 2
##  This test adds a new keyword to a header and compares the result to
## an image of expected results. Because the hedit task works on images
## in-place, a copy of the input image is created before running hedit,
## in order to preserve the original image.
##
-->

<RegTest>
<pre-exec>
   <command>import shutil</command>
   <command>shutil.copy('m51.fits', 'm51_test2.fits')</command>
</pre-exec>

<title>images/imutil/hedit: test 2</title>
<level>2</level>
<taskname>hedit</taskname>
<pfile>hedit_test2.par</pfile>
<output>
  <val>
    <file>m51_test2.fits</file>
    <reference>m51_test2_ref.fits</reference>
    <comparator>image</comparator>
    <ignorekeys>filename,date,iraf-tlm</ignorekeys>
  </val>
</output>

</RegTest>
```

Translate to python:

```
# To reproduce the regtest functionality, use these imports:

import pandokia.helpers.process as process
import pandokia.helpers.filecomp as filecomp


#
# The old xml tests contain a single test in a <regtest> block.
# We write that as a test in python.  The most convenient way to do
# it is as a test function.  This works with both nose and py.test
#

def test_2():

    # Even though the test is named after the test function, we
    # still need a string for the test name.  We use it later.
    testname = 'test_2'

    # You need to have a tda dict for:
    #  - recording information to make FlagOK work
    #  - recording parameters to the task as attributes
    global tda
    tda = { }
```

(continues on next page)

(continued from previous page)

```
    # We use the same information from the <output> section, but
    # written as a python data structure instead of xml.
    #  <output>
    #    <val>
    #      <file>m51_test2.fits</file>
    #      <reference>m51_test2_ref.fits</reference>
    #      <comparator>image</comparator>
    #      <ignorekeys>filename,date,iraf-tlm</ignorekeys>
    #    </val>
    #  </output>

    output = [
            # one dict for each output file to compare (i.e. each <val>)
            {
                # copy file, reference, and comparator from the XML
                'file'      : 'm51_test2.fits',
                'reference' : 'm51_test2_ref.fits',
                'comparator': 'image',

                # any other XML fields in the <val> should be put in a
                # dict and stored in args:
                'args'      : {
                    # fitsdiff args ignorekeys and ignorecomm take
                    # a list instead of comma separated text
                    'ignorekeys': [ 'filename', 'date', 'iraf-tlm' ],
                    'maxdiff' : .0001,
                 },

            },
            # if there are more files, list more dicts here
        ]

    # delete all the output files before starting the test
    filecomp.delete_output_files( output )

    # for <pre-exec> and <post-exec>, just write the python commands
    # directly into your test
    #
    # <pre-exec>
    # <command>import shutil</command>
    # <command>shutil.copy('m51.fits', 'm51_test2.fits')</command>
    # </pre-exec>
    #
    import shutil
    shutil.copy('m51.fits', 'm51_test2.fits')

    # <title> and <level> don't count for anything in pandokia,
    # so ignore them

    # The old regtest runner loads the IRAF tasks for tables,
    # stsdas, images, and fitsio.  Load as many of the IRAF tasks
    # as you need.
    #
    # Skip this section if your test does not require pyraf/iraf.
    import pyraf        # not all tests need or want pyraf
    from pyraf.iraf import tables
    from pyraf.iraf import stsdas
```

(continues on next page)

(continued from previous page)

```
    # for an IRAF task, use this helper function to run it
    #
    # <taskname>hedit</taskname>
    # <pfile>hedit_test2.par</pfile>
    #
    process.run_pyraf_task( 'hedit', 'hedit_test2.par', tda=tda )

    # If you have a post-exec, put the code here

    # compare the output files - use this exact command
    filecomp.compare_files( output, ( __file__, testname ), tda = tda,)
```

9.3 Python: Using JSON or pprint to compare complex data structures

abstract JSON or pprint can convert various complex data structures into printable text for user display. Your tests can convert your data to a printable format and diff the results to make a comparison.

We have various systems that contain nested trees of lists and dicts. You can display these data items with python builtins.

JSON makes a very readable format, but it does not represent every possible python object. pprint makes a slightly less readable format, but knows (for example) the difference between tuples and lists.

Using json:

```
import json

# to a string
s = json.dumps( mydata, indent=4, sort_keys=True, default=str )

# to a file
f=open("out/myfile.txt","w")
json.dump( mydata, f, indent=4, sort_keys=True, default=str )
```

pprint understands python objects better, but is harder to read.

Using pprint:

```
import pprint

# to a string
s = pprint.pformat( mydata, indent=4 )

# to a file
f=open("out/myfile.txt","w")
pprint.pprint( mydata, stream=f, indent=4 )
```

Once you have the object in a string or file, you can use various diff-like tools to compare it.

This example uses difflib and json. The reference data is a string constant in the source code.

```
import json
import pandokia.helpers.filecomp as filecomp
```

(continues on next page)

(continued from previous page)

```
l = [ { 'a' : 1, 'b' : 2 }, [ 1, 2 ] ]
result = json.dumps( l, indent=4, sort_keys=True, default=str )
print result

ref = """
[
    {
        "a": 1,
        "b": 2
    },
    [
        1,
        2
    ]
]
"""
assert filecomp.diffjson( result, ref )
```

This example creates an output file and compares to a reference file. You can update the reference file by using the okify feature in the gui.

```
import os.path
import json
import pandokia.helpers.filecomp as filecomp

def test_1() :
    global tda
    tda = { }

    # list of files to compare
    files = [
            ( "test_1.txt", "diff", { 'rstrip' : True } )
        ]

    # delete output files before running test
    filecomp.delete_output_files( files )

    # some data
    l = [ { 'a' : 1, 'b' : 2 }, [ 1, 2 ] ]

    # make the output file ( in directory out/ )
    f = open("out/test_1.txt","w" )
    result = json.dump( l, f, indent=4, sort_keys=True, default=str )
    f.close()

    # file comparison tool
    filecomp.compare_files(
        # files to compare
        files,
        # name to use to construct okfile
        "okfile/" + os.path.basename(__file__) + ".test_1",
        # tda dict (to record name of okfile)
        tda
        )
```

9.4 Python: Implementing foo.test() for your package

abstract This is an example of how to make a test() function in your package. This test function can be called directly by a user, or it can be used as part of a pandokia test run.

This section describes various patterns for tests that are installed with your package and called with a function named test(). For example:

```
% python
Python 2.7.1 (r271:86832, Jan  7 2011, 09:41:02)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import xxx
>>> xxx.test()
>>> xxx.test(verbose=True)
```

To run it from the shell, you can use the command:

```
python -c 'import sys, xxx; sys.exit(xxx.test())'
```

All of the patterns described here provide the same interface, but not all of them do anything different for verbose=True.

The test() function can be used in a pandokia test run by executing the test() function from some file that pandokia runs.

Using the "run" runner:

In pdk_runners, add the line:

```
mytest.py   run
```

In mytest.py, use:

```
#!/usr/bin/env python
import xxx
xxx.test()
```

pdkrun will see mytest.py, recognize it as a "run"-type file, and execute the python in that file. It just runs the file, not using py.test/nose/whatever, but your test() function will use one of these test frameworks, and therefore will be able to report results into the pandokia system.

9.4.1 using py.test

In your package, create a subsidiary package named "tests" (note the 's' in the name). The tests package should contain python files that will be recognized by py.test.

In your __init__.py:

```
def test( verbose=False ) :
    #
    import os, pytest

    # find the directory where the test package lives
    from . import tests
    dir = os.path.dirname( tests.__file__ )

    # assemble the py.test args
```

(continues on next page)

(continued from previous page)

```
    args = [ dir ]

    # run py.test
    try :
        return pytest.main( args )
    except SystemExit as e :
        return e.code
```

Install your tests in xxx.tests with file names that py.test will recognize.

9.4.2 using nose

In your package, create a subsidiary package named "tests" (note the 's' in the name). The tests package should contain python files that will be recognized by nose.

In your __init__.py:

```
def test( verbose=False ) :
    import os, nose

    # find the directory where the test package lives
    from . import tests
    dir = os.path.dirname( tests.__file__ )

    # get the name of the test package
    argv = [ 'nosetests', '--exe', dir ]

    # run nose
    try :
        return nose.main( argv = argv )
    except SystemExit as e :
        return e.code
```

Install your tests in xxx.tests with file names that nose will recognize.

9.4.3 using pycode

In your package, create a subsidiary package named "tests" (note the 's' in the name). The tests package should contain modules that implement your tests.

Here is a sample package has tests in packagename.tests.test_a and packagename.tests.test_b .

Place this in __init__.py:

```
def test( verbose=False ) :
    import pandokia.helpers.pycode as pycode

    return pycode.package_test(
        parent = __name__,
        test_package = 'tests',
        test_modules = [ 'test_a', 'test_b' ],
        verbose = verbose,
    )
```

Write your tests in packagename/tests/test_a.py as:

```
import pandokia.helpers.pycode as pycode

with pycode.test('some_thing') as t :
    assert some_thing

with pycode.test('it_works') as t :
    assert it_works
```

When writing pycode tests using the with-statement, you can nest tests:

```
with pycode.test('top') as tt :
    setup()

    with pycode.test('mid') as tm :
        more_setup()

        assert something
        # this assert reports a test named "top.mid"

        with pycode.test('bottom') as tb :
            assert something_else
            # this test is named "top.mid.bottom"
```

See . . . for details.

9.4.4 using multiple runners

If you have tests written for multiple test frameworks, you can have your test function invoke each of the frameworks separately. For example:

```
def test_pytest( verbose=False ) :
    # ... as in examples above,
    # ... but using tests.pytest for the test package

def test_nose( verbose=False ) :
    # ... as above
    # ... but using tests.nose for the test package

def test_pycode( verbose=False ) :
    # ... as above
    # ... but using tests.pycode for the test package

def test( verbose=False ) :
    pt = test_pytest(verbose)
    no = test_nose(verbose)
    pc = test_pycode(verbose)
    return pt | no | pc
```

Of course, this means that you need more than one test framework installed to run all the tests. This is an incovenience to the user, who may have to install all three of pandokia, py.test and nose to run all the tests.

It could be useful during a transition period, especially if you structure the various test functions to be aware of whether they can run or not:

```
def test_pytest( verbose=False ) :
    try :
```

(continues on next page)

(continued from previous page)

```
        import pytest
    except ImportError :
        print "Cannot import pytest - pytest tests are skipped"
        return
    ...
```

There is an example of this usage in test_new/pdkrun_test_data/test_fn in the pandokia source code (https://svn.stsci.edu/svn/ssb/etal/pandokia/trunk/test_new/pdkrun_test_data/test_fn).

9.5 Python: Running external executables in your test

There are some helper functions for running external programs. You can use these instead of any of the half-dozen or so variants of popen that are in the standard library. This interface is simpler, it will not be deprecated as often as the other popen variants, and it is easier to be compatible with python testing packages.

9.5.1 Do not use subprocess

You most likely want to capture the stdout and stderr of the external program in your test log. By default, this will not work in subprocess – it will go to the stdout/stderr of pdkrun. If you want to use subprocess, you have to take special actions to capture the stdout/stderr and repeat it into Python's stdout/stderr.

Instead, use the run_process function:

```
# NO
import subprocess
subprocess.call( [ 'ls', '-l' ] )

# YES
import pandokia.helpers.process as process

# run the program, capturing output into a file
status = process.run_process( [ 'ls', '-l' ], output_file='myfile.tmp' )

# read the file and print it to the stdout being used by the test
process.cat( [ "myfile.tmp" ] )
```

Since Pandokia will only run one test at a time in any working directory, it is not necessary to guarantee a unique name for the temp file. Since the output file will be created, it is not necessary to delete it before using it.

9.5.2 Running an external process

`run_process` executes a command with stdout/stderr redirected into a file:

```
import pandokia.helpers.process as pr
import os

arg = [ 'ls', '-ld', '.' ]

# run the process with arg list, using default environment and output file
status = pr.run_process( arg )

# explicitly read the output
```

(continues on next page)

(continued from previous page)

```
out = open('output_file','r').read()
print "output was:"
print out
print "exit code is %d" % status

# run the process, explicitly specify everything
status = pr.run_process( arg, env=os.environ, output_file='myfile.tmp' )
out = open('myfile.tmp','r').read()

# just repeat the output to our own stdout
print "output was:"
pr.cat( [ 'myfile.tmp' ] )
print "exit code is %d" % status
```

`cat` reads each file in a list and writes the content to stdout:

```
import pandokia.helpers.process as pr
pr.cat( [ 'myfile.tmp' ] )
```

This is much like `os.system("cat myfile.tmp")` with one significant difference: os.system will write stdout to the same stdout that pandokia is writing to, so the output will not appear in your test report; `process.cat` will write the output into the sys.stdout that the test is using, which means that the file content will appear in the test report.

9.5.3 IRAF tasks

If you have Pyraf installed, you can run IRAF tasks:

```
import pandokia.helpers.filecomp as filecomp
import pandokia.helpers.process as process

tda = { }

f = open('parfile','w')
f.write('''
input,s,a,"dev$pix",,,Input images
output,s,a,"foo.fits",,,Output images or directory
verbose,b,h,no,,,Print operations performed?
mode,s,h,ql
''')
f.close()

filecomp.safe_rm('foo.fits')

process.run_pyraf_task('imcopy', 'parfile', output_file='foo.txt', tda=tda )
```

The tda dict is optional. If you supply it, task parameters will be entered as TDAs.

This is a simplified interface for pyraf tasks. You can also use the full capabilities of pyraf directly.

If the task returns an error, it will raise pr.RunPyrafTaskException which contains the values `task` and `err`:

```
except process.RunPyrafTaskException as e :
    print "task ",e.task
    print "err  ",e.err
```

The call may also raise any of the pyraf-related exceptions.

9.6 Python: Parameterized Tests

abstract When you have a list and you want to run the same test function for each item in the list.

9.6.1 minipyt

```
import pandokia.helpers.pycode as pycode

p_list = [
     ( 'nameA', 'v1', 'v2' ),
     ( 'nameB', 'va', 'vc' ),
     ]

for name, v1, v2 in p_list :
    with pycode.test(name) :
        assert foo(v1, v2)
```

The tests will be named `nameA` and `nameB`

9.6.2 py.test

```
import pytest

p_list = [
     ( 'nameA', 'v1', 'v2' ),
     ( 'nameB', 'va', 'vc' ),
     ]

@pytest.mark.parametrize( ('name', 'v1', 'v2'), p_list)
def test_thing(name, v1, v2) :
    assert foo(v1,v2)
```

The tests will be named `test_thing[nameA-v1-v2]` and `test_thing[nameB-va-vc]`

Take notice that the spelling of "parametrize" does not contain an "e" before the "r".

9.6.3 nose

```
p_list = [
     ( 'nameA', 'v1', 'v2' ),
     ( 'nameB', 'va', 'vc' ),
     ]

def run_it( v1, v2 ) :
    assert foo(v1,v2)

def test_thing() :
    for name, v1, v2 in p_list :
        yield run_it, v1, v2
```

The tests will be named `test_thing('v1',_'v2')` and `test_thing('va',_'vc')`

CHAPTER 10

Adding Test Runners to Pandokia

abstract When Pandokia executes tests, it uses a test runner to execute the set of tests in a specific file. You can add interfaces to your own test runners, as long as they can report their results in pandokia format.

This document describes how to support a new test runner.

Contents

10.1 Pick a name

Your test runner needs a name that pandokia can use internally. In the examples, I will add a test runner named *shell_runner*.

10.2 Describe file names for that kind of test

Pandokia uses a wild card pattern to recognize all the files that belong to a particular test_runner.

If you are adding a new runner to the pandokia source tree, change the value of runner_glob in pandokia/runners/__init__.py

If you are adding your own runner, change the value of runner_glob in the installed module pandokia.config

Choose your wildcard and add it to the list of patterns:

```
runner_glob = [
#    ( 'not_a_test*.py',     None         ),
    ( '*.py',               'nose'       ),
    ( '*.xml',              'regtest'    ),
    ( 'test*.sh',           'shell_runner' ),
]
```

10.3 Define the python code to interface with your runner

For a runner named XYZ, you must create either the module *pandokia.runners.XYZ* (to build your runner into pandokia) or *pandokia_runner_XYZ* (to create your runner as a separately installable python module).

For example, for *shell_runner*, you would create the file *pandokia/runners/shell_runner.py*.

In this file, you define the functions that Pandokia will call to locate your runner.

The functions are:

- def command(env)

  The function *command* returns a command that pandokia should use to execute a test. The parameters are all in the dictionary *env*, along with other environment variables. The returned command is used roughly like this:

  ```
  p = subprocess.Popen(cmd, shell=True, env = env )
  ```

  (for exact details, see the function run() in pandokia/run_file.py)

  Normally, we expect that you will want to run your testing system in a separate process. This allows the test system to keep operating, even if something about a particular test or test_runner causes a crash.

  Instead of a single command, this function can return a list of commands to be executed in order.

- def lst(env)

  The function *list* returns a list of the test names that are in the file, but does not execute any of the tests. When a test file is disabled, this feature is used to find a list of disabled tests to include in the pandokia log file.

  It is not always convenient to implement this feature. If it is not, return None.

The parameter *env* is a dictionary of environment variables that would be used to execute the test. Everything you need to know is stored in this environment.

The specific variables of interest are:

- PDK_DIRECTORY

  The name of the directory that the test is in. Pandokia always runs the test from the directory where the test is found, so this value is the same as os.getcwd()

- PDK_FILE

  The name of the file that contains the tests. The file is in the current working directory.

- PDK_LOG

Your test runner should append test results to this file.

- PDK_TESTRUN

  The name of the test run that this test execution is part of.

- PDK_PROJECT

  The name of the project that this test execution is part of.

- PDK_CONTEXT

  The name of the context that this test is running in.

- PDK_TESTPREFIX

  This prefix represents the directories at higher levels in the directory tree. If the prefix is not '', you should insert the prefix and a '/' in front of the test name.

- PDK_PARALLEL
- PDK_PROCESS_SLOT

  Internal tracking values used when executing tests in parallel. These values are not directly useful to a test_runner, but the system does not remove them from the environment.

Other environment variables are also present, either from the environment inherited from your shell or from the pdk_environment files.

10.4 Implement the command that runs your tests

You must provide a program that actually runs the tests. It should use arguments and/or environment variables to know what to do.

You should APPEND data in pandokia log format to the file named in $PDK_LOG.

See doc/file_format.txt and doc/report_fields.txt for details of the report format.

Before starting your program, pdkrun wrote some default values to the log file. These are:

```
test_run
project
host
location
test_runner
context
```

At a minimum, you must add:

```
test_name
status
END
```

You may report values that override the defaults, and you may add other fields as described in doc/report_fields.txt.

10.5 Using pycode.report in your python-base test runner

If you are writing in python, you can use the "reporter" object in pandokia.helpers.pycode to write properly formatted records to $PDK_LOG :

```
import pandokia.helpers.pycode as pycode

# initialize one instance of the pycode reporter; if you are
# running in pandokia, you don't need any parameters except
# the None.
rpt = pycode.reporter( None )

# declare your test name and a dict for the attributes.
test_name = 'some_test'
tda = { }
tra = { }

# start the test.  tda will not be used until the call to finish()
# so more tdas can still be added by the test code.
# this call remembers what time the test started.
rpt.start( test_name, tda )

# cause python to redirect sys.stdout and sys.stderr into a StringIO
pycode.snarf_stdout()

# perform the test.  It should fill in attributes in tda[]
# and tra[] and set the value of status.
foo()

# capture the redirected stdout
log = pycode.end_snarf_stdout()

# report the result of the currently running test.  This call
# knows what time the test finished.
rpt.finish( status, tra, log )

rpt.close()
```

You can also report the test all at once instead of splitting into start()/finish() :

```
import pandokia.helpers.pycode as pycode

# initialize one instance of the pycode reporter; if you are
# running in pandokia, you don't need any parameters except
# the None.
rpt = pycode.reporter( None )

# declare your test name and a dict for the attributes.
test_name = 'some_test'
tda = { }
tra = { }

start_time = time.time(0)

# cause python to redirect sys.stdout and sys.stderr into a StringIO
pycode.snarf_stdout()

# perform the test.  It should fill in attributes in tda[]
# and tra[] and set the value of status.
foo()

# capture the redirected stdout
```

(continues on next page)

(continued from previous page)

```
log = pycode.end_snarf_stdout()

# report the result of the test.  Leave out optional args
# if you don't want to report them.
rpt.report( test_name, status, start_time = start_time,
    end_time = time.time(0), tra = tra, tda = tda, log = log )

rpt.close()
```

This is a primitive tool for writing log files. Calls to rpt.start() and rpt.finish() do not nest, and attempts to write to the same file with more than one rpt object are likely to end badly.

CHAPTER 11

Pandokia Database Administration

11.1 Initial Setup

11.2 Running Tests

TODO: fix this to know the current state of the implementation

- XX_latest , XX_yesterday recognizes if XX is in cfg.recurring_prefix
- common.looks_like_a_date() to show day names
- common.run_previous(), common.run_next()

See doc/pdk_aware.txt for details on running tests. The result of running a test with pdkrun is one or more pdk log files. You somehow need to get those to your pandokia server where you can import them into the database with the 'pdk import' command.

There is a special set of test run names that follow the pattern 'daily_YYYY-MM-DD' where Y/M/D are the year/month/day of the test. These names get special treatment in the display software. We run that test run each day and automatically import the results into our system.

For example, last night it ran 'daily_2009-08-17'. I can query this through the web interface either by that name or by 'daily_latest', which is converted to today's date internally.

If you run daily tests (e.g. for continuous integration), it is helpful to use this naming convention.

11.3 Importing Test Results

When you have you pdk log files on the server host, use:

```
pdk import filename
```

to import each one into the database.

At STScI, many machines write their pdk log files to a common directory on a file server. On the pdk server host, we have a cron job that is essentially

```
cd /server/that_directory
pdk import *
```

to import the data after all the tests are run.

When a test result is already in the database, it prints a warning. If you do not complete an import, you can run it again; you just get a lot of warnings about records that already exist.

11.4 Expected / Missing Tests

The database keeps a list of expected tests according to a class of test runs.

In our system, the morning cron job imports the results from the previous night's tests, then executes these commands:

```
pdk gen_expected daily daily_latest
```

In this example, "daily" is the class of tests and "daily_latest" is translated into the name of last night's test run.

You can have as many types of expected test as you like, and you can update the expected list from whatever test run you like. For example:

```
pdk gen_expected marks_tests mark-25
```

would find all the tests currently listed for test run mark-25 and add them to the "marks_tests" class. In this operation, expected tests are added only, never removed. If you run gen_expected several times, you get the union of all the tests identified.

Once you have a class of expected tests defined, you can check if all of them are present in a test run:

```
pdk check_expected daily daily_latest
```

finds any "daily" tests that are not present in the latest daily test run:

```
pdk check_expected marks_tests mark-26
```

checks that mark-26 contains all of the expected marks_tests tests. Any that are missing will be entered in the database with status M (for missing).

If a missing test is imported later, the imported data replaces the M record.

When a test is no longer expected, you can remove that expectation in two ways:

- When viewing a list of tests in the web UI, you can select tests and use the "Not Expected" button.
- You can remove records from the database table directly.

```
sqlite3 /where/ever/pdk.db
DELETE FROM expected WHERE test_run_type = 'whatever';
```

11.5 Importing Contacts

Contacts are handled separately from the test results. On any *one* of the test machines:

```
pdk_gen_contact projectname /where/ever/it/is > file.contact

scp file.contact my_pandokia_server:
```

then on the server, run:

```
pdk import_contact < file.contact
```

This adds contacts only. To remove contacts, delete records from the table "contact". For example, you can delete all contacts with the SQL command:

```
DELETE FROM contact;
```

and then import them all again.

11.6 Emailed Announcements

This section TBD.

11.7 Deleting Old Test Data

When you have test runs to delete from the database:

11.7.1 Delete the primary records

```
pdk delete -test_run daily_2009-03-10
    # deletes just that one day's results
```

You can use "*" as a wild card if it is at the beginning or end of a name:

```
pdk delete -test_run 'user_xyzzy_2012-07-*'
```

You can use SQL wild cards:

```
pdk delete -test_run 'user_xyzzy_2012-07-%'
```

You can specify multiple parameters to delete only a portion of a test run. It will delete only that portion that matches all the parameters listed:

```
pdk delete -test_run 'user_xyzzy_*' -project 'pyetc' -context 'trunk'
    -host 'etcbrady' -status 'M'
```

This will not delete records that are part of any test run marked "valuable".

11.7.2 Delete secondary records

The initial delete only removes enough of the data to make the test no longer appear on the reports. There is a second step that is not performed at this stage because it is much slower.

Clean up related records:

```
pdk clean
```

Because of the large volume (easily many millions of records for a single day's test runs), this step can take a long time. Instead of requiring this to happen during *pdk delete*, we provide it as a separate step.

pdk clean does the delete in groups of a few hundred tests at a time. You can interrupt it whenever you get tired of waiting, then restart it again later.

It is not necessary to run the clean step every time you delete records. In a normal system, an administrator will run the clean step from time to time.

11.7.3 Notes

- If you will delete several test runs, it is convenient to 'pdk delete' each of them, then use a single *pdk clean* command afterwards. We commonly allow anyone to run *pdk delete* alone, then run a scheduled *pdk clean* during off hours.
- The database files do not necessarily get smaller when you delete data, but space in the file is available to be re-used.
- 'pdk clean' does a lot of work. In sqlite, it tries not to keep the database locked for too long, but that is hard to achieve. If using sqlite, it is best to run it when the database is not otherwise busy.

11.7.4 STScI Routine Database Cleaning

pandokia.cfg.pdk_db.table_usage() returns the current database size as best it can for the database you are using. Part of our continuous integration system uses this call to generate a report when the database exceeds our current limit.

When it does, we have somebody identify the oldest daily test runs in the database and delete them with a command like

```
pdk delete -test_run 'daily_2012-10-%'
```

We repeat that command for each type of daily test runs (we have several), and then run

```
pdk clean
```

We usually delete a month at a time. Because of our high test volume (several million records per month), both of these steps take a very long time.

11.8 Deleting Old QID data

The system stores data relating to some queries in the database. You should clean this out now and then with just:

```
pdk clean_queries
```

11.9 Sample Nightly Scripts

These sample scripts give you an idea of how we use Pandokia. There are a set of coordinated cron jobs that run our tests overnight:

on each test machine:

```
cd /where/my/tests/are
proj=my_project
testrun=daily_`date '+%Y-%m-%d'`
logname=/fileserver/pdk_logs/$hostname.$testrun
pdk run -parallel 4 -log $logname -test_run $testrun -project $proj -r .
```

on the server:

```
cd /fileserver/pdk_logs
mkdir -p old
pdk import /fileserver/pdk_logs/*
mv * old

pdk gen_expected   daily daily_latest
pdk check_expected daily daily_latest

pdk email daily_latest
```

Of course, we also have scripts that first install the software to be tested.

11.10 Some Database Notes

Here are some database notes:

11.10.1 Mysql

```
mysql -p
create database pandokia;
use pandokia;

drop database pandokia;
```

```
set password [ for user ] = password("xyzzy") ;
```

```
use mysql;
update user set password=PASSWORD("xyzzy") where user = 'dude' ;
flush privileges;
```

CHAPTER 12

Appendix: Glossary

Meta-Runner: the program invoked when the "pdk run" command is given. The meta-runner discovers tests, sets up the test environment, and invokes the appropriate test runner(s) to run tests.

PDK Log: an ascii file containing a series of test results. See file_format.rst.

TDA: Test Definition Attribute. Test authors may define TDAs to associate information about the test input or properties with the test result.

TRA: Test Result Attribute. Test authors may define TRAs to associate more detailed information about the test output than the simple status.

Test result: a complete record describing the result of a test containing all required fields. A test result is represented in various ways: it is written in a log file (by a combination of default and test-specific fields), and it is stored in the database. See report_fields.rst.

Test runner: a specific test runner, such as nose.

CHAPTER 13

Appendix: Importing Demo Data

TODO: THIS CHAPTER IS PROBABLY OUT OF DATE

After you have installed Pandokia on your system, you can follow this procedure to import the provided demo data into your database.

13.1 About the demo data

In the sample_data directory, you will find a series of pdk log files with various names. These files were created by:

- running a set of tests, which we designated as test run demo_1, on a computer named banana

  pdkrun –parallel 4 –project sample –test_run=demo_1 –log sample_data/demo_1.banana -r tests

- running the same test run on a computer named justine

  pdkrun –parallel 4 –project sample –test_run=demo_1 –log sample_data/demo_1.justine -r tests

- running the same set of tests, but designated as test run demo_2, on banana

  pdkrun –parallel 4 –project sample –test_run=demo_2 –log sample_data/demo_2.banana -r tests

These are both four-processor machines, so we ran the tests in 4 parallel processes to make best use of the hardware, and each process produced its own pdk log file, with a distinguishing numerical suffix. We designated this the "sample" project for convenience.

13.2 Importing the demo data

All the data can be imported with one command:

```
cd sample_data
pdk import *
```

or you can import them one test run at a time:

```
pdk import demo_1.*
pdk import demo_2.*
```

The importer prints some diagnostic information to stdout as it processes each file.

13.3 Browsing the demo data

Point your browser at the top level pandokia webpage (if you followed the instructions in INSTALL/"Using the pandokia test web server", then this will be URL localhost:7070/pdk.cgi)

At this point you can navigate from the top level to the test run and host that you're interested in. The figure (report_flow.png, report_flow_caption.png) illustrates the possible navigation paths.

In this case, we'll start by clicking in the "Lists of Test Runs" on "All". Both demo test runs show up, with several active links:

- if you click on the name of the test run, you get a tabular view

  Here the tests are grouped by host and status. Clicking on any of the links will take you to a page showing only those tests that satisfy the relevant conditions.

- clicking on "T" gives you the treewalker view

  Here the tests are grouped by status and the first element of the hierarchical test name. One can click on any of the links in the table which will again take you to a page showing only those tests that satisfy the relevant conditions, or use one of the "narrow to" choices to further narrow by host or project.

- clicking on "P" gives you a summary of only the problem cases (status = error or fail)

  This is a summary view, pre-selected to show only the tests with an error or failing status. From here you can click on an individual test name to see its test result; or "detail of all" down at the bottom to see them all on one page.

  From a summary view, you can compare to another test run by typing its name into the box and clicking "compare". This will show the status of each test from the "other" run (ie, the one whose name you typed in) next to the status from this run. You can also click "different", which will show only the tests whose status changed between the two runs (none in this case), or "same" for the complementary set.

  You can also "add attributes" to a summary view. This will expand the table to include one column for every TDA or TRA that was defined by any of the tests in the set. (Thus, it may be a sparse table, if the tests defined different attributes.) That's not very interesting in the demo test set, but for real tests, it might provide useful clues about what failing tests had in common, or further detail about how badly a test failed.

CHAPTER 14

FAQ

= Pandokia FAQ =

== General ==

- Why did you write Pandokia? What is it for?

  I used to come to work and find an email message listing all the tests that failed last night, but it is hard to work with just a list of the names of hundreds of failed tests. (We run tens of thousands of tests, distributed across many operating systems; even a tiny change to one part of the system can cause many tests to fail.)

  Pandokia is a reporting system to organize all those results. Of course, you can't view a report until you have data, so it also includes a mechanism to run a testing tool (such as py.test) and gather the test results.

  Having Pandokia has enabled us to increase our automatic testing substantially. Some nights, we run over 150 000 tests, and we can easily manage the case where thousands of tests fail.

- Why "pandokia"? Does it mean anything?

  We made up the word from the Greek morphemes "pan" = all and "dokimi" = test.

- Where can I get help?

  You can send email to help@stsci.edu: you must put "STSDAS/pandokia" in the message for the system to route it to us. We also follow the TIP list (testing-in-python@idyll.org).

== Design ==

- Why didn't you use Zope, Django, Ruby on Rails, ASP.NET, etc?

  Tools like that can be useful, but when you have a lot of work to do, sometimes it is best to use something you already know instead of evaluating a bunch of new tools. I recommend that software developers should read the short story "Superiority" by Arthur C Clarke.

  The simple CGI implementation is very easy to install on a web server, without root privileges.

- Why did you invent your own file format for pdk log files instead of using XML, JSON, YAML, INI, etc?

It is easier to read and easier to write than most standard formats. We even create pdk log files from shell scripts.

If a test crashes while part way through writing a pandokia log file, another test can come along and append to that log file without any data corruption; this is not true for many standard formats.

In the pandokia format, the report from the crashed test will not be complete, but the reports from the tests that follow are still readable. This is important because core dumps are a fact of life when you use continuous integration.

- What databases are supported?
  - sqlite

    Sqlite does not require a database server. We initially set up a database without getting the IT department involved.

  - MySQL

    When we wanted more granular locking than sqlite,

  - Postgres - maybe

    I have postgres on my home computer, and I have run pandokia there from time to time. I don't routinely test in postgres.

- Why not use an ORM ?

  I know SQL fairly well for a causal database user. For me, object-relation managers are hard to use, but SQL is easy.

== Trivia ==

- Why is there a cat on the top of the web displays?

  We chose to use a cat shortly after an overdose of goats at the testing BOF at PyCon 2010.

  This cat is Vicki's cat, Sienna. She is alertly watching your test reports, as you should be.

- But what about the goat?

  1. You can change the file pandokia/head.png to a png file of whatever image you would like.
  2. If you just need to see a goat, we recommend: http://en.wikipedia.org/wiki/File:ZodiacalConstellationCapricornus.jpg

CHAPTER 15

Programming in Pandokia

abstract This is stuff you want to know if you are working on the Pandokia software.

15.1 Database Programming in Pandokia

abstract Pandokia uses a SQL-based database. It uses it directly through DBAPI, not through an ORM. There are some conventions to follow to implement more portable SQL.

Contents

- *EXPLAIN QUERY*

15.1.1 DBAPI limitations

DBAPI defines something that looks like a standard interface to databases, but you can't quite write a program assuming DBAPI and expect it to work with any database that offers a DBAPI interface.

One big problem comes in parameter passing. If you look at PEP 249 (http://www.python.org/dev/peps/pep-0249/), you can see 5 possible values for paramstyle. None of them are available in every database.

Notably, mysqldb and pyscopg2 offer only 'format' and 'pyformat', while sqlite3 offers 'qmark', 'named', and some parameter formats that are not part of DBAPI.

Solution

The solution is an SQL execute that converts a standard format for parameters to whatever the database engine wants.

cursor = pdk_db.execute(statement, parameters)

> perform a database action with named parameters
>
> statement contains an instance of :AAA for each named parameter. The parameter name has to match the regular expression pattern [a-zA-Z0-9_]+
>
> If parameters is a dict, it will be used as-is.
>
> If parameters is a list or tuple, it will be converted into a dict with keys '1', '2', '3', . . . , so you can write your query using :1, :2, :3, etc.
>
> If parameters is any other type, it is an error.
>
> It IS NOT permitted to have the character ':' in your sql, even if it occurs inside a string literal. The library does not check for this, but will likely choke on your query.
>
> It IS NOT permitted to have the character '%' in your sql, even if it occurs inside a string literal. This limitation is inherited from some of the DBAPI implementations. You may be able to get away with it, depending which DBAPI you are using, but your code will not be portable.
>
> Notably, you cannot use LIKE 'xxx%' but you can use LIKE :1 and pass a parameter of "xxx%".

The return value is a cursor

Why this interface?

It is easy and fast for regex substition to convert this to something that any dbapi database can use.

I would like your sql to be allowed to say " where a like 'arf%' ", but the % will not correctly pass through some of the dbapi implementations. Since this interface is intended to be portable, you can't have the %. You have to do " where a like :1 " and pass 'arf%' as a parameter.

Stylistically, I like ":arf" better than "%(arf)s"

Some people like ORMs, but I find them substantially harder to use than regular SQL. Some queries that are easy in SQL are difficult or impossible in some ORMs.

What about performance?

This interface takes a small amount of additional time for each query, but the performance reduction from using an interpreted language like Python is so large that I don't notice (or care) about the difference. The portability is more important.

This interface has not been benchmarked against common ORM implementations such as those available in SqlAlchemy or Django, but those systems use much more complex methods for constructing queries. I conjecture that any performance difference favors this interface, but that the total difference is insignificant.

15.1.2 Connecting to the databse

Within pandokia

```
import pandokia
pdb_db = pandokia.cfg.pdk_db

cursor = pdk_db.execute( query, parameters )
```

Always fully specify the columns to retrieve; never use "SELECT *".

Use :1, :2, . . . for parameters when you have only a fixed set of parameters.

```
c = pdk_db.execute("select a, b from tbl where a = :1 and b = :2",('a_value','b_value
↪'))
for x in c :
    print c[0],c[1]
```

Without pandokia

- using MySQLdb:

```
import pandokia.db_mysql

db = pandokia.db_mysql.PandokiaDB( access_arg )
    # access_arg is the same as you would use with MySQLdb
```

- using sqlite3 or pysqlite2:

```
import pandokia.db_sqlite

db = pandokia.db_sqlite.PandokiaDB( filename )
    # filename is the same as you would use with sqlite3
```

- using psyscopg2 (postgresql):

```
import pandokia.db_psycopg2

db = pandokia.db_psycopg2.PandokiaDB( access_arg )
    # access_arg is the same as you would use with psycopg2
```

- using pymssql (Microsoft SQL Server):

```
import pandokia.db_pymssql

db = pandokia.db_pymssql.PandokiaDB( access_arg )
    # access_arg is the same as you would use with pymssql
```

The object does not connect to the database when you create it. You can call db.open() to explicitly connect, or it will connect to the database the first time it needs the connection.

Without pandokia, if you have a Django settings.py module

```
# hook up to the database
import pandokia.db as dbm
import pyetc.etc_web.settings as settings

db = dbm.db_from_django( settings )
```

This works for mysql and sqlite.

The object does not connect to the database when you create it. You can call db.open() to explicitly connect, or it will connect to the database the first time it needs the connection.

15.1.3 Schemas

If you use database-specfic features in your schema, you just have to write a separate schema for each database engine.

There are a few significant differences in schemas for different databases:

- sqlite databases allow VARCHAR without a length, but others do not.
- Different databases use different approaches to autoincrementing columns. See result_scalar.key_id in pandokia/sql/*.sql to see the different approaches.
- Some databases do not have auto-increment columns. This abstraction layer cannot hide that for you.
- The details of what indexes you want may vary between database implementations.

This lacks the "magic" of an ORM automatically generating your schema, but is not so bad if you have a small number of tables or a small number of databases.

15.1.4 Dynamically constructed WHERE clauses

where_dict is a function to dynamically construct WHERE clauses, based on a list of column names and values.

The parameter to where_dict is a list of (column_name, value), where column_name is a required column name and value is a value to match. All the columns are ANDed together. If the value for a column is a list, the possible values are ORed together.

The value may contain "*x", "x*", or "*x*", which will be converted to "%x", "x%", or "%x%" and used in a LIKE clause. Other glob-like characters are not permitted.

If the value contains '%', it will be used in a LIKE clause.

The '_' character does NOT automatically create a LIKE expression because it is too common in our data values, but note that "A_B*" will translate to LIKE 'A_B%'

There is no good way to search for values containing *, %, [, or ?

Example:

```
where_text, where_dict = pdk_db.where_dict( [
    ( 'a', 1 ),
    ( 'b', [ 'x', 'y' ] ),
    ( 'c', 'z*' )
    ] )

c = pdk_db.execute("SELECT a,b FROM tb %s"%where_text, where_dict)
```

is equivalent to

```
where_text = "WHERE ( a = :1 ) AND ( b = :2 OR b = :3 ) AND ( c LIKE :4 )"
where_dict = {
    '1' : 1,
    '2' : 'x',
    '3' : 'y',
    '4' : 'z%'
    }
c = pdk_db.execute("SELECT a,b FROM tb %s"%where_text, where_dict)
```

15.1.5 COMMIT / ROLLBACK

Commit and rollback work the same as with dbapi; use the pandokia database object:

```
pdk_db.commit()

pdk_db.rollback()
```

15.1.6 Exceptions

IntegrityError happens when you violate a database constraint.

```
db = xxx.PandokiaDB( args )

try :
    c = db.execute('INSERT INTO ...')
except db.IntegrityError as e :
    ...
```

ProgrammingError is a problem such as a syntax error in your SQL.

```
try :
    c = db.execute('...')
except db.ProgrammingError as e :
    ...
```

DBAPI implementations can raise other exceptions that are not yet implemented by the pandokia interface.

Postgres will raise an exception if you get an error from one SQL statement and to not rollback() before executing more statements.

15.1.7 Table Usage

You can ask the database for the amount of space used by the data. There is not always a clear answer to this question, but this function returns the best available answer in a database specific way:

```
i = db.table_usage()
print "using %d bytes"%i
```

In mysql, this is the sum of the table and index sizes from “SHOW TABLE STATUS”.

In sqlite3, this is the size of the database file.

15.1.8 EXPLAIN QUERY

You can get a description of how the database will evaluate the query with:

```
s = pdk_db.explain_query( text, qhere_dict )
print s
```

This is highly database dependent.

CHAPTER 16

Environment variables

These environment variables are used by pandokia:

PDK_CONTEXT

As input to pdkrun: equivalent to –context

As input to a runner: The name of the context to be reported for the tests to be run.

PDK_DIRECTORY

As input to a runner: The full path name of the current directory where the test is executing.

PDK_FILE

As input to a runner: The name of the file that contains tests to run in this process. pandokia runners may use this to know which file to run tests from, though a runner may also be written to take the file name as a parameter.

PDK_LOG

As input to pdkrun: equivalent to –log

As input to a runner: The name of the file to append PDK_LOG entries to. This is how the test runner reports results to the rest of the system.

PDK_PARALLEL (input, output)

As input to pdkrun: equivalent to –parallel ; the number of concurrent test runners that may execute.

As input to a runner: The max number of concurrent test runners that may be executing. Not particularly useful.

PDK_PROCESS_SLOT

As input to a runner: A small integer that uniquely identifies one of the concurrent test processes. You can use this for unique temp file names.

PDK_PROJECT

As input to pdkrun: equivalent to –project

As input to a runner: The name of the project to be reported for the tests to be run.

PDK_STATUSFILE

As input to pdkrun or "pdk status": Name of file to record currently executing processes.

PDK_TESTPREFIX

As input to a runner: This MUST be prepended to the local test names that the runner reports. It contains the location in the hierarchy where the current test file is located.

PDK_TESTRUN

As input to pdkrun: equivalent to –test_run ; the name of the test run to report.

As input to a runner: The name of the test_run to be reported for the tests to be run.

PDK_TIMEOUT

As input to pdkrun: The number of wall clock seconds that a test runner may be allowed to run. This time is per-file. Processes that exceed this limit will be killed, first with SIGTERM and 10 seconds later with SIGKILL if necessary. Processes that survive SIGKILL for more than 10 seconds will be assumed to be wedged and ignored.

PDK_TMP

Used internally to locate certain temp files.

CHAPTER 17

Assorted Tools

Pandokia contains some programming tools that were convenient to dump into this project. These are not necessarily related to testing.

Contents

17.1 pdk_sphinxweb - automatically build several sphinx documents

pdk_sphinxweb searches for sphinx documents nested below the current directory. A directory might contain a sphinx document if one of these conditions is true:

- it is named “doc”
- it is named “docs”
- it contains a file named “.this_is_a_sphinx_doc”

In order for the document to be built, the directory must not contain a file named “no_auto_build” and must contain a file named “Makefile”.

Invoke pdk_sphinxweb with the name of the directory where you want the built documents installed:

```
pdk_sphinxweb /x/y/z/output_directory
```

It will find things that look like sphinx documents in the **current** directory, attempt to build them as HTML and PDF, and place the results in the output directory. When finished, it will construct index.shtml with document titles, links to HTML and PDF documents, and links to HTML and PDF build logs.

If the output_directory already contains header.html or footer.html, those files will be included in the index.shtml file. Otherwise, you get a bare table, which any reasonable web browser will be able to display.

17.2 xtname - set the title of an xterm

Use:

```
xtname this is a window title
```

to set the title of an xterm from your shell prompt.

17.3 tbconv - simple text table conversion tool

Pandokia contains a table generator that it uses to produce the tables on the web page. tbconv reads standard input in one of a few different table formats then displays the table on standard output in one of a few different table formats.

input formats:

- csv - CSV files; uses the python csv module
- rst - reStructuredText; only knows Simple tables, not Grid tables
- tabs - columns are separated by tab characters
- trac_wiki - table format used by track wiki (can also specify as “tw”)

output formats are the same as input formats, plus:

- html - HTML

17.4 sendto - subversion branching tool

tbd

17.5 junittopdk - convert JUnit/XML output to Pandokia format

tbd

CHAPTER 18

Indices and tables

- genindex
- search

Index