pytest-wdl

Release 1.4.1

John Didion, Michael T. Neylon

Nov 17, 2020

CONTENTS:

1 } **5**

2 } **7**

3 } **9**

4 } **11**

5 } **15**

6 } **17**

7 } **21**

8 } **23**

9) **27**

10) **29**

11) **31**

12 pytest_wdl **33**

12.1 pytest_wdl package 33

13 Indices and tables **57**

Python Module Index **59**

Index **61**

User manual

pytest-wdl is a plugin for the [pytest](https://docs.pytest.org/en/latest/) unit testing framework that enables testing of workflows written in [Workflow Description Language](https://github.com/openwdl). Test workflow inputs and expected outputs are [configured](#test-data) in a *test_data.json* file. Workflows are run by one or more [executors](#executors). By default, actual and expected outputs are compared by MD5 hash, but data type-specific comparisons are provided. Data types and executors are pluggable and can be provided via third-party packages.

Dependencies

- Python 3.6+
- **At least one of the supported workflow engines (see [Limitations](#known-limitations) for known limitations of these work**
  - [Miniwdl](https://github.com/chanzuckerberg/miniwdl) (v0.6.4 is automatically installed as a dependency of pytest-wdl)
  - [Cromwell](https://github.com/broadinstitute/cromwell/releases/) JAR file (pytest-wdl is currently tested with Cromwell v53.1)
  - [dxWDL](https://github.com/dnanexus/dxWDL) JAR file (pytest-wdl is currently tested with dxWDL v1.42)
- Java-based workflow engines (e.g. Cromwell and dxWDL) require a Java runtime (typically 1.8+)
- If your WDL tasks depend on Docker images, make sure to have the [Docker](https://www.docker.com/get-started) daemon running

Other python dependencies are installed when you install the library.

Installation

Install from PyPI

`commandline $ pip install pytest-wdl `

Install from source

You can to clone the repository and install:

` $ make install `

Or use pip to install from github:

`commandline $ pip install git+https://github.com/elililllyco/pytest-wdl.git `

Install optional dependencies

Some optional features of pytest-wdl have additional dependencies that are loaded on-demand.

The plugins that have extra dependencies are:

- dx: Support for DNAnexus file storage, and for the dxWDL executor.

- bam: More intelligent comparison of expected and actual BAM file outputs of a workflow than just comparing MD5 checksums.
- http: Support for executors that use HTTPS protocol to communicate with a remote server (e.g. Cromwell Server)
- yaml: Support using YAML for configuration and test data files. Note that *.yaml* files are ignored if a *.json* file with the same prefix is present.
- progress: Show progress bars when downloading remote files.

To install a plugin's dependencies:

` $ pip install pytest-wdl[<plugin>] `

To do this locally, you can clone the repo and run:

`commandline $ pip install -e .[<data_type>] `

To install pytest-wdl and **all** extras dependencies:

` $ pip install pytest-wdl[all] `

Quick Start

The full code for this quick-start project is available in the [examples](https://github.com/EliLillyCo/pytest-wdl/tree/develop/examples/quickstart) directory.

To demonstrate how to use pytest-wdl, we will create a simple variant calling workflow and tests with the following project structure:

```
quickstart |_variant_caller.wdl |_tests
    |_data | |_NA12878.chr22.tiny.vcf |_test_data.json |_test_variant_caller.py
```

Our workflow is shown below. It requires a BAM file and a reference sequence, it calls a task to perform variant calling using [Freebayes](https://github.com/ekg/freebayes), and it returns a VCF file with the called variants.

```wdl
version 1.0

struct Reference {
  File fasta
  String organism
}

workflow call_variants {
    input {
      File bam
      Reference ref
    }
    call freebayes {
        input:
          bam=bam, index=ref
    }
    output {
      File vcf = freebayes.vcf
    }
}

task freebayes {...}
```

Now you want to test that your workflow runs successfully. You also want to test that your workflow always produces the same output with a given input. You can accomplish both of these objectives with the following test function, which is in the *tests/test_variant_caller.py* file:

```python def test_variant_caller(workflow_data, workflow_runner):
    inputs = { "bam": workflow_data["bam"], "ref": {
            "fasta": workflow_data["reference_fa"], "organism": "human"
        }
    } expected = workflow_data.get_dict("vcf") workflow_runner(
        "variant_caller.wdl", inputs, expected
    )
```

This test executes a workflow with the specified inputs, and will compare the outputs to the specified expected outputs. The *workflow_data* and *workflow_runner* parameters are [fixtures](https://docs.pytest.org/en/latest/fixture.html) that are injected by the pytest framework at runtime. The *workflow_data* fixture provides the test data files based on the following configuration in the *tests/test_data.json* file. Note that some of the data files are stored remotely (in this case, in the [GitHub repository for FreeBayes](https://github.com/ekg/freebayes/tree/65251646f8dd9c60e3db276e3c6386936b7cf60b/test/tiny)) and are downloaded automatically when needed. Later in the mnaual you'll find [descriptions of these fixtures](#fixtures) and details on how to [configure](#configuration) your tests' inputs, exected outputs, and other parameters.

```json {
    "bam": { "url": "https://.../NA12878.chr22.tiny.bam"
    }, "reference_fa": {
        "url": "https://.../q.fa"
    }, "vcf": {
        "name": "NA12878.chr22.tiny.vcf", "type": "vcf"
    }
```

CHAPTER ONE

}

To run this test, make sure you have pytest-wdl [installed](#installation) and run the following command from within the project folder. The default executor (miniwdl) will be used to execute the workflow. You can configure any of the other supported executors by creating a [configuration file](#configuration).

` $ pytest -s -vv --show-capture=all `

This test should run successfully, and you should see output that looks like this:

Project setup

pytest-wdl should support most project set-ups, including:

```` # simple myproject |_workflow.wdl |_subworkflow1.wdl |_subworkflow2.wdl |_tests

|_test_workflow.py |_test_data.json

# multi-module with single test directory myproject |_main_workflow.wdl |_module1 | |_module1.wdl |_module2 | |_module2.wdl |_tests

|_main | |_test_main.py | |_test_data.json |_module1

|_test_module1.py |_test_data.json

...

# multi-module with separate test directories myproject |_main.wdl |_module1 | |_module1.wdl | |_tests | |_test_module1.py | |_test_data.json |_module2 | |_... |_tests

|_test_main.py |_test_data.json

```

By default, pytest-wdl tries to find the files it is expecting relative to one of two directories:

- Project root: the base directory of the project. In the above examples, *myproject* is the project root directory. By default, the project root is discovered by looking for key files (e.g. setup.py), starting from the directory in which pytest is executing the current test. In most cases, the project root will be the same for all tests executed within a project.
- **Test context directory: starting from the directory in which pytest is executing the current test, the test context directory i**
  - In the "simple" and "multi-module with single test directory" examples, *myproject* would be the test context directory
  - In the "multi-module with separate test directories" example, the test context directory would be *myproject* when executing *myproject/tests/test_main.py* and *module1* when executing *myproject/module1/tests/test_module1.py*.

## Fixtures

All functionality of pytest-wdl is provided via fixtures. As long as pytest-wdl is in your *PYTHONPATH*, its fixtures will be discovered and made available when you run pytest.

The two most important fixtures are:

- [workflow_data](#test_data):+ Provides access to data files for use as inputs to a workflow, and for comparing to workflow output.
- [workflow_runner](#executors): Given a WDL workflow, inputs, and expected outputs, runs the workflow using one or more executors and compares actual and expected outputs.

There are also [several additional fixtures](#configuration) used for configuration of the two main fixtures. In most cases, the default values returned by these fixtures "just work." However, if you need to override the defaults, you may do so either directly within your test modules, or in a [conftest.py](https://docs.pytest.org/en/2.7.3/plugins.html) file.

## Test data

Typically, workflows require inputs and generate outputs. Beyond simply ensuring that a workflow runs successfully, we often want to additionally test that it reproducibly generates the same results given the same inputs.

Test inputs and outputs are configured in a *test_data.json* file (or *test_data.yaml* file if you have the [YAML](#yaml) dependency installed) that is stored in the same directory as the test module. This file has one entry for each input/output. Primitive types map as expected from JSON to Python to WDL. Object types (e.g. structs) have a special syntax. For example, the following *test_data.json* file defines an integer input that is loaded as a Python *int* and then maps to the WDL *Integer* type when passed as an input parameter to a workflow, and an object type that is loaded as a Python dict and then maps to a user-defined type (struct) in WDL:

```json {
    "input_int": 42, "input_obj": {
        "class": "Person", "value": {
            "name": "Joe", "age": 42
        }
    }
```

# CHAPTER TWO

# }

### Files

For file inputs and outputs, pytest-wdl offers several different options. Test data files may be located remotely (identified by a URL), located within the test directory (using the folder hierarchy established by the [datadir-ng](https://pypi.org/project/pytest-datadir-ng/) plugin), located at an arbitrary local path, or defined by specifying the file contents directly within the JSON file. Files that do not already exist locally are localized on-demand and stored in the [cache directory](#configuration-file).

Some additional options are available only for expected outputs, in order to specify how they should be compared to the actual outputs.

File data can be defined the same as object data (i.e. "file" is a special class of object type):

```json {
    **"config": {** "class": "file", "value": {
        "path": "config.json"
    }
}
```

CHAPTER

THREE

}

As a short-cut, the “class” attribute can be omitted and the map describing the file provided directly as the value. Below is an example *test_data.json* file that demonstrates different ways to define input and output data files:

```json {
    **“bam”: {** “url”: “http://example.com/my.bam”, “http_headers”: {
        “auth_token”: “TOKEN”
    }, “digests”: {
        “md5”: “8db3048a86e16a08d2d8341d1c72fecb”
    }
}, “reference”: {
    “path”: “${REFERENCE_DIR}/chr22.fa”
}, “sample”: {
    “path”: “samples.vcf”, “contents”: “sample1nsample2”
}, “output_vcf”: {
    “name”: “output.vcf”, “type”: {
        “name”: “vcf”, “allowed_diff_lines”: 2
    }
}
```

# CHAPTER FOUR

}

The available keys for configuring file inputs/outputs are:

- *name*: Filename to use when localizing the file; when none of *url*, *path*, or *contents* are defined, *name* is also used to search for the data file within the tests directory, using the same directory structure defined by the [datadir-ng](https://pypi.org/project/pytest-datadir-ng/) fixture.
- *path*: The local path to the file. If the path does not already exist, the file will be localized to this path. Typically, this is defined as a relative path that will be prefixed with the [cache directory](#configuration-file) path. Environment variables can be used to enable the user to configure an environment-specific path.
- *env*: The name of an environment variable in which to look up the local path of the file.
- ***url*: A URL that can be resolved by [urllib](https://docs.python.org/3/library/urllib.html).**
  - ***http_headers*: Optional dict mapping header names to values. These headers are used for file download requests.**
    - *env*: The name of an environment variable in which to look up the header value.
    - *value*: The header value; only used if an environment variable is not specified or is unset.
- *contents*: The contents of the file, specified as a string. The file is written to *path* the first time it is requested.
- *digests*: Optional mapping of hash algorithm name to digest. These are digests that have been computed on the remote file and are used to validate the downloaded file. Currently only used for files resolved from URLs.

In addition, the following keys are recognized for output files only:

- *type*: The file type. This is optional and only needs to be provided for certain types of files that are handled specially for the sake of comparison. The value can also be a hash with required key "name" and any other comparison-related attributes (including data type-specific attributes).
- *allowed_diff_lines*: Optional and only used for outputs comparison. If '0' or not specified, it is assumed that the expected and actual outputs are identical.

#### URL Schemes

pytest_wdl uses *urllib*, which by default supports http, https, and ftp. If you need to support alternate URL schemes, you can do so via a [plugin](#plugins). Currently, the following plugins are avaiable:

- *dx* (DNAnexus): requires the *dxpy* module

#### Data Types

When comparing actual and expected outputs, the "type" of the expected output is used to determine how the files are compared. If no type is specified, then the type is assumed to be "default."

##### default

The default type if one is not specified.

- It can handle raw text files, as well as gzip compressed files.
- If *allowed_diff_lines* is 0 or not specified, then the files are compared by their MD5 hashes.
- If *allowed_diff_lines* is > 0, the files are converted to text and compared using the linux *diff* tool.

##### vcf

- During comparison, headers are ignored, as are the QUAL, INFO, and FORMAT columns; for sample columns, only the first sample column is compared between files, and only the genotype values for that sample.
- **Optional attributes:**
  - *compare_phase*: Whether to compare genotype phase; defaults to False.

##### bam*:

- BAM is converted to SAM.
- Replaces random UNSET-w*b type IDs that samtools often adds.
- One comparison is performed using all rows and a subset of columns that are expected to be invariate. Rows are sorted by name and then by flag.
- A second comparison is performed using all columns and a subset of rows based on filtering criteria. Rows are sorted by coordinate and then by name.
- **Optional attributes:**
  - *min_mapq*: The minimum MAPQ when filtering rows for the second comparison.
  - *compare_tag_columns*: Whether to include tag columns (12+) when comparing all columns in the second comparison.

* requires extra dependencies to be installed, see [Install Optional Dependencies](#install-optional-dependencies)

## Executors

An Executor is a wrapper around a WDL workflow execution engine that prepares inputs, runs the tool, captures outputs, and handles errors. Currently, the following executors are supported (but aternative executors can be implemented as [plugins](#plugins)):

- **[Cromwell](https://cromwell.readthedocs.io/)**
  - Local: run Cromwell locally (*executor="cromwell"*)
  - Server: run Cromwell via a remote Cromwell server instance (*executor="cromwell-server"*)
- [Miniwdl](https://github.com/chanzuckerberg/miniwdl): run miniwdl locally in the same process as pytest-wdl (*executor="miniwdl"*)
- [dxWDL](https://github.com/dnanexus/dxWDL): compile tasks/workflows using dxWDL and run them remotely on DNAnexus (*executor="dxwdl"*)

The *workflow_runner* fixture is a callable that runs the workflow using the executor.

The first argument to *workflow_runner* is the path to the WDL file. It is required. It may be an absolute or a relative path; if the latter, it is first searched relative to the current *tests* directory (i.e. *test_context_dir/tests*), and then the project root.

The remaining arguments to *workflow_runner* are optional:

- *inputs*: Dict that will be serialized to JSON and provided to the executor as the workflow inputs. If not specified, the workflow must not have any required inputs.

- *expected*: Dict mapping output parameter names to expected values. Any workflow outputs that are not specified are ignored. This parameter can be omitted if, for example, you only want to test that the workflow completes successfully.
- *workflow_name*: The name of the workflow to execute in the WDL script. If not specified, the name of the workflow is extracted from the WDL file.
- *inputs_file*: Specify the inputs.json file to use, or the path to the inputs.json file to write, instead of a temp file.
- *imports_file*: Specify the imports file to use. By default, all WDL files under the test context directory are imported if an *import_paths.txt* file is not provided.
- *executors*: Optional list of executors to use to run the workflow. Each executor will be run in a [subtest](https://github.com/pytest-dev/pytest-subtests). If not specified the [default_executors](#configuration) are used.
- *callback*: An optional function that is called after each successful workflow execution. Take three arguments:

  ```python from pathlib import Path

  **def callback(executor_name: str, execution_dir: Path, outputs: dict):** ...

  ```

You can also pass executor-specific keyword arguments.

### Executor-specific *workflow_runner* arguments

#### Cromwell Local

- *imports_file*: Instead of specifying the imports file, this can also be the path to the imports zip file to write, instead of a temp file.
- *java_args*: Override the default Java arguments.
- *cromwell_args*: Override the default Cromwell arguments.

#### Cromwell Server

- *imports_file*: Instead of specifying the imports file, this can also be the path to the imports zip file to write, instead of a temp file.
- *timeout*: Number of seconds to allow a workflow to run before timing out with an error; defaults to 3600 (1hr).

#### Miniwdl

- *task_name*: Name of the task to run, e.g. for a WDL file that does not have a workflow. This takes precedence over *workflow_name*.

#### dxWDL

Requires the *dxpy* package to be installed.

<!–* *task_name*: Name of the task to run, e.g. for a WDL file that does not have a workflow. This takes precedence over *workflow_name*.–> * *project_id*: ID of the project where the workflow will be built. Defaults to the currently selected project. You can also specify different projects for workflows and data using *workflow_project_id* and *data_project_id* varaibles. * *folder*: The folder within the project where the workflow will be built. Defaults to '/'. You can also specify different folders for workflows and data using *workflow_folder_id* and *data_folder_id* varaibles. * *stage_id*: Stage ID to use when inputs don't come prefixed with the stage. Defaults to "stage-common". * *force*: Boolean; whether to force the workflow to be built even if the WDL file has not changed since the last build. * *archive*: Boolean; whether to archive existing applets/workflows (True) or overwrite them (False) when building the workflow. Defaults to True. * *extras*: [Extras file](https://github.com/dnanexus/dxWDL/blob/master/doc/ExpertOptions.md) to use when building the workflow.

### Known limitations

#### Cromwell

- Cromwell issues are tracked in the [Broad's Jira](https://broadworkbench.atlassian.net/projects/BA/issues).

#### miniwdl

- **pytest-wdl is currently pinned to [miniwdl version 0.6.4](https://github.com/chanzuckerberg/miniwdl/releases/tag/v0.6.4),**
    - Task input files are mounted read-only by default; commands to rename or remove them can succeed only with –copy-input-files
- See the [miniwdl open issues](https://github.com/chanzuckerberg/miniwdl/issues) for other potential limitations

#### dxWDL

- DNAnexus (and thus the dxWDL executor) does not support optional collection types (e.g. *Array[String]?*, *Map[String, File]?*).
- See the [dxWDL open issues](https://github.com/dnanexus/dxWDL/issues) for other potential limitations

## Writing tests in JSON

Rather than writing test cases in Python, in most instances it is possible to define your tests completely in JSON (or in YAML, if you have the [YAML](#yaml) dependency installed). Python- and JSON-based tests can co-exist.

JSON tests are defined in a file that starts with *test* and ends with *.json*. There is one required top-level key, *"test"*, whose value is an array of hashes. Each hash defines a single test.

```json {
    "tests": [
        { "name": "mytest", "wdl": "mytest.wdl", "inputs": {
            "input_files": [ "test_file1", "test_file2"
            ], "organism": "human"
        }, "expected": {
            "output_file": "expected_file"
        }
    }
]
```

# CHAPTER FIVE

## }

A test has two required keys:

- *name*: The test name; must be unique within the file.
- *wdl*: A path to a WDL file (equivalent to the first parameter to *workflow_runner* described above).

Any of the other parameters to *workflow_runner* can be specified as keys as well. To refer to *workflow_data* entries, simply use the entry key as a value. For example, "test_file1", "test_file2", and "expected_file" in the example above are defined in the *test_data.json* file. If a string value is not found in the *workflow_data*, it is treated as a string literal. For example, the value of the "organism" key ("human") is treated as a string literal because there is no "human" key defined in *test_data.json*.

Instead of using a separate *test_data.json* file, you may instead put your test data definitions directly in the test JSON file as the value of the "data" key. Note that it is either/or - if you put your test data definitions in your tests JSON file, then any definitions in *test_data.json* will not be ignored.

```json {
    "data": {
        "test_file": {  "url": "https://foo.com/input.txt"
        }, "expected_file": {
            "url": "https://foo.com/output.txt"
        }
    }, "tests": [
        {  "name": "mytest", "wdl": "mytest.wdl", "inputs": {
                "input_file": "test_file", "organism": "human"
            }, "expected": {
                "output_file": "expected_file"
            }
        }
    ]
```

CHAPTER

SIX

}

## Configuration

pytest-wdl has two levels of configuration:

- Project-specific configuration, which generally deals with the structure of the project, and may require customization if the structure of your project differs substantially from what is expected, but also encompases executor-specific configuration.
- Environment-specific configuration, which generally deals with idiosyncrasies of the local environment.

### Project-specific configuration

Configuration at the project level is handled by overriding fixtures, either in the test module or in a top-level conftest.py file. The following fixtures may be overridden:

<table border="1" class="docutils"> <thead> <tr> <th>fixture name</th> <th>scope</th> <th>description</th> <th>default</th> </tr> </thead> <tbody> <tr> <td><code>project_root_files</code></td> <td>module</td> <td>List of filenames that are found in the project root directory.</td> <td><code>["setup.py", "pyproject.toml", ".git"]</code></td> </tr> <tr> <td><code>project_root</code></td> <td>module</td> <td>The root directory of the project. Relative paths are relative to this directory, unless specified otherwise.</td> <td>Starting in the current test directory/module, scan up the directory hierarchy until one of the <code>project_root_files</code> are located.</td> </tr> <tr> <td><code>workflow_data_descriptor_file</code></td> <td>module</td> <td>Path to the JSON file that describes the test data.</td> <td><code>tests/test_data.json</code></td> </tr> <tr> <td><code>workflow_data_descriptors</code></td> <td>module</td> <td>Mapping of workflow input/output names to values (as described in the <a href="#files">Files</a> section).</td> <td>Loaded from the <code>workflow_data_descriptor_file</code></td> </tr> <tr> <td><code>workflow_data_resolver</code></td> <td>module</td> <td>Provides the <code>DataResolver</code> object that resolves test data; this should only need to be overridden for testing/debugging purposes</td> <td><code>DataResolver</code> created from <code>workflow_data_descriptors</code></td> </tr> <tr> <td><code>import_paths</code></td> <td>module</td> <td>Provides the path to the file that lists the directories from which to import WDL dependencies</td> <td>"import_paths.txt"</td> </tr> <tr> <td><code>import_dirs</code></td> <td>module</td> <td>Provides the directories from which to import WDL dependencies</td> <td>Loaded from <code>import_paths</code> file, if any, otherwise all WDL files under the current test context directory are imported</td> </tr> <tr> <td><code>default_executors</code></td> <td>session</td> <td>Specify the default set of executors to use when running tests</td> <td><code>user_config.default_executors</code></td> </tr> </tbody> </table>

### Environment-specific configuration

There are several aspects of pytest-wdl that can be configured to the local environment, for example to enable the same tests to run both on a user's development machine and in a continuous integration environment.

Environment-specific configuration is specified either or both of two places: a JSON configuration file and environment variables. Environment variables always take precendence over values in the configuration file. Keep in mind that (on a *nix system) environment variables can be set (semi-)permanently (using *export*) or temporarily (using *env*):

```commandline # Set environment variable durably $ export FOO=bar
```

# Set environment variable only in the context of a single command $ env FOO=bar echo "foo is $FOO" ```

#### Configuration file

The pytest-wdl configuration file is a JSON-format file. Its default location is *$HOME/.pytest_wdl_config.json* (or *$HOME/.pytest_wdl_config.yaml* if you have the [YAML](#yaml) dependency installed). To get started, you can copy one of the following example config files and modify as necessary:

- [simple](https://github.com/EliLillyCo/pytest-wdl/blob/develop/examples/config/simple.pytest_wdl_config.json): Uses only the miniwdl executor
- [more complex](https://github.com/EliLillyCo/pytest-wdl/blob/develop/examples/config/complex.pytest_wdl_config.json): Uses both miniwdl and Cromwell; shows how to configure proxies and headers for accessing remote data files in a private repository

The available configuration options are listed in the following table:

<table border="1" class="docutils"> <thead> <tr> <th>configuration file key</th> <th>environment variable</th> <th>description</th> <th>default</th> <th>recommendation</th> </tr> </thead> <tbody> <tr> <td><code>cache_dir</code></td> <td><code>PYTEST_WDL_CACHE_DIR</code></td> <td>Directory to use for localizing test data files.</td> <td>Temporary directory; a separate directory is used for each test module</td> <td>pro: saves time when multiple tests rely on the same test data files; con: can cause conflicts, if tests use different files with the same name</td> </tr> <tr> <td><code>execution_dir</code></td> <td><code>PYTEST_WDL_EXECUTION_DIR</code></td> <td>Directory in which tests are executed</td> <td>Temporary directory; a separate directory is used for each test function</td> <td>Only use for debugging; use an absolute path</td> </tr> <tr> <td><code>proxies</code></td> <td>Configurable</td> <td>Proxy server information; see details below</td> <td>None</td> <td>Use environment variable(s) to configure your proxy server(s), if any</td> </tr> <tr> <td><code>http_headers</code></td> <td>Configurable</td> <td>HTTP header configuration that applies to all URLs matching a given pattern; see details below</td> <td>None</td> <td>Configure headers by URL pattern; configure headers for specific URLs in the test_data.json file</td> </tr> <tr> <td><code>show_progress</code></td> <td>N/A</td> <td>Whether to show progress bars when downloading files</td> <td>False</td> <td></td> </tr> <tr> <td><code>default_executors</code></td> <td>PYTEST_WDL_EXECUTORS</td> <td>Comma-delimited list of executor names to run by default</td> <td>["cromwell"]</td> <td></td> </tr> <tr> <td><code>executors</code></td> <td>Executor-dependent</td> <td>Configuration options specific to each executor; see below</td> <td>None</td> <td></td> </tr> <tr> <td><code>providers</code></td> <td>Provider-dependent</td> <td>Configuration options specific to each provider; see below</td> <td>None</td> <td></td> </tr> <tr> <td>N/A</td> <td><code>LOGLEVEL</code></td> <td>Level of detail to log; can set to 'DEBUG', 'INFO', 'WARNING', or 'ERROR'</td> <td>'WARNING'</td> <td>Use 'DEBUG' when developing plugins/fixtures/etc., otherwise 'WARNING'</td> </tr> </tbody> </table>

##### Proxies

In the proxies section of the configuration file, you can define the proxy servers for schemes used in data file URLs. The keys are scheme names and the values are either strings - environment variable names - or mappings with the following keys:

- *env*: The name of an environment variable in which to look for the proxy server address.
- *value*: The value to use for the proxy server address, if the environment variable is not defined or is unset.

```json {

    **"proxies": {**

        **"http": {** "env": "HTTP_PROXY"

        }, "https": {

            "value": "https://foo.com/proxy", "env": "HTTPS_PROXY"

        }
```

```
}
```

# CHAPTER SEVEN

}

##### HTTP(S) Headers

In the http_headers section of the configuration file, you can define a list of headers to use when downloading data files. In addition to *env* and *value* keys (which are interpreted the same as for [proxies](#proxies), two additional keys are allowed:

- *name*: Required; the header name
- *pattern*: A regular expression used to match the URL; if not specified, the header is used with all URLs.

```json {
    **"http_headers": [**
        { "name": "X-JFrog-Art-Api", "pattern": "http://my.company.com/artifactory/*", "env": "TOKEN"
        }
    ]
```

CHAPTER

# EIGHT

}

##### Executor-specific configuration

###### Java-based Executors

These options apply to all Java-based executors (currently Cromwell and dxWDL):

<table border="1" class="docutils"> <thead> <tr> <th>configuration file key</th> <th>environment variable</th> <th>description</th> <th>default</th> </tr> </thead> <tbody> <tr> <td><code>java_bin</code></td> <td><code>JAVA_HOME</code></td> <td>Path to java executable; If not specified, then Java executable is expected to be in $JAVA_HOME/bin/java</td> <td>None</td> </tr> <tr> <td><code>java_args</code></td> <td><code>JAVA_ARGS</code></td> <td>Arguments to add to the <code>java</code> command</td> <td><code>-Dconfig.file=<cromwell_config_file></code> (for Cromwell executor, if <code>cromwell_config_file</code> is specified)</td> </tr> <tr> <td>N/A</td> <td><code>CLASSPATH</code></td> <td>Java classpath; searched for a file matching "cromwell*.jar" if <code>cromwell_jar</code> is not specified</td> <td>None</td> </tr> </tbody> </table>

###### Cromwell Local

<table border="1" class="docutils"> <thead> <tr> <th>configuration file key</th> <th>environment variable</th> <th>description</th> <th>default</th> </tr> </thead> <tbody> <tr> <td><code>cromwell_jar_file</code></td> <td><code>CROMWELL_JAR</code></td> <td>Path to Cromwell JAR file</td> <td>None</td> </tr> <tr> <td><code>cromwell_configuration</code></td> <td><code>CROMWELL_CONFIG</code></td> <td>Path to Cromwell configuration file</td> <td>None</td> </tr> <tr> <td><code>cromwell_args</code></td> <td><code>CROMWELL_ARGS</code></td> <td>Arguments to add to the <code>cromwell run</code> command</td> <td>None;</td> </tr> </tbody> </table>

Note that if you are doing your development locally and using Docker images you've built yourself, it is recommended to add *-Ddocker.hash-lookup.enabled=false* to *java_args* to disable Docker lookup by hash. Otherwise, you must push your Docker image(s) to a remote repository (e.g. DockerHub) before running your tests.

###### Cromwell Server

- *cromwell_api_url*: The full path to the cromwell API (i.e. http://localhost:8000/api/workflows/v1).
- *cromwell_api_username*: The username to authenticate against the cromwell api if protected
- *cromwell_api_password*: The password to authenticate against the cromwell api if protected
- *cromwell_configuration*: Configuration (file or dict) to pass to cromwell when submitting run requests.

###### dxWDL

The dxWDL executor (as well as URLs using the *dx:*// scheme) require you to be logged into DNAnexus. You can configure either a username and password or an auth token in the config file to log in automatically (see [provider configuration](#dnanexus)), otherwise you will be asked to log in interactively.

<table border="1" class="docutils"> <thead> <tr> <th>configuration file key</th> <th>environment variable</th> <th>description</th> <th>default</th> </tr> </thead> <tbody> <tr> <td><code>dxwdl_jar_file</code></td>

<td><code>DXWDL_JAR</code></td> <td>Path to dxWDL JAR file</td> <td>None</td> </tr> <tr> <td><code>dxwdl_cache_dir</code></td> <td><code>DXWDL_CACHE_DIR</code></td> <td>Directory to use to cache downloaded results</td> <td>A temporary directory is used and deleted after each test</td> </tr> </tbody> </table>

##### Provider-specific configuration

A "provider" is a remote (generally cloud-based) service that provides both an execution engine and data storage.

###### DNAnexus

<table border="1" class="docutils"> <thead> <tr> <th>configuration file key</th> <th>environment variable</th> <th>description</th> <th>default</th> </tr> </thead> <tbody> <tr> <td><code>dx_username</code></td> <td>None</td> <td>Username to use for logging into DNAnexus if the user is not already logged in</td> <td>None</td> </tr> <tr> <td><code>dx_password</code></td> <td>None</td> <td>Password to use for logging into DNAnexus if the user is not already logged in</td> <td>None</td> </tr> <tr> <td><code>dx_token</code></td> <td>None</td> <td>Token to use for logging into DNAnexus if the user is not already logged in (mutually exclusive with username/password</td> <td>None</td> </tr> </tbody> </table>

##### Fixtures

There are two fixtures that control the loading of the user configuration:

<table border="1" class="docutils"> <thead> <tr> <th>fixture name</th> <th>scope</th> <th>description</th> <th>default</th> </tr> </thead> <tbody> <tr> <td><code>user_config_file</code></td> <td>session</td> <td>The location of the user configuration file</td> <td>The value of the <code>PYTEST_WDL_CONFIG</code> environment variable if set, otherwise <code>$HOME/.pytest_wdl_config.json</code></td> </tr> <tr> <td><code>user_config</code></td> <td>session</td> <td>Provides a <code>UserConfiguration</code> object that is used by other fixtures to access configuration values</td> <td>Default values are loaded from <code>user_config_file</code>, but most values can be overridden via environment variables (see <a href="#configuration">Configuration</a>)</td> </tr> </tbody> </table>

## Plugins

pytest-wdl provides the ability to implement plugins for data types, executors, and url schemes. When two plugins with the same name are present, the third-party plugin takes precedence over the built-in plugin (however, if there are two conflicting third-party plugins, an exception is raised).

### Creating new data types

To create a new data type plugin, add a module in the *data_types* package of pytest-wdl, or create it in your own 3rd party package.

Your plugin must subclass the *pytest_wdl.data_types.DataFile* class and override its methods for *_assert_contents_equal()* and/or *_diff()* to define the behavior for this file type.

Next, add an entry point in setup.py. If the data type requires more dependencies to be installed, make sure to use a *try/except ImportError* and raise a PluginError with an informative message, and to add the extra dependencies under the setup.py's *extras_require*. For example:

```python # plugin.py from pytest_wdl.plugins import PluginError

**try:** import mylib

**except ImportError as err:**

    **raise PluginError(** "mytype is not available because the mylib library is not " "installed"

    ) from err

```

```python from setuptools import setup
```

```
setup( ..., entry_points={
        "pytest_wdl.data_types": [ "mydata = pytest_wdl.data_types.mytype:MyDataFile"
        ]
    }, extras_require={
        "mydata": ["mylib"]
    }
```

CHAPTER

NINE

)

In this example, the extra dependencies can be installed with *pip install pytest-wdl[mydata]*.

### Creating new executors

To create a new executor, add a module in the *executors* package, or in your own 3rd party package.

Your plugin must subclass *pytest_wdl.executors.Executor* and implement an initalizer and the *run_workflow()* method. The initalizer must take *import_dirs: Optional[Sequence[Path]]* as its first argument, and may take additional executor-specific keyword arguments.

```python from pathlib import Path from pytest_wdl.executors import Executor from typing import Optional, Sequence

**class MyExecutor(Executor):**

**def __init__(** self, import_dirs: Optional[Sequence[Path]], myarg: str = "hello"

**):** ...

**def run_workflow(** self, wdl_path: Path, inputs: Optional[dict] = None, expected: Optional[dict] = None, **kwargs

**) -> dict:** ...

```

Next, add an entry point in setup.py. If the executor requires more dependencies to be installed, make sure to use a *try/except ImportError* to warn about this and add the extra dependencies under the setup.py's *extras_require* (see example under [Creating new data types](#creating-new-data-types)). For example:

```python from setuptools import setup

**setup(** ..., entry_points={

**"pytest_wdl.executors": [** "myexec = pytest_wdl.executors.myexec:MyExecutor"

]

}, extras_require={

"myexec": ["mylib"]

}
```

# CHAPTER TEN

# )

### Supporting alternative URL schemes

If you want to use test data files that are available via a service that does not support http/https/ftp downloads, you can implement a custom URL scheme.

Your plugin must subclass *pytest_wdl.url_schemes.UrlScheme* and implement the *scheme*, *handles*, and any of the *urlopen*, *request*, and *response* methods that are required.

Next, add an entry point in setup.py. If the schem requires more dependencies to be installed, make sure to use a *try/except ImportError* to warn about this and add the extra dependencies under the setup.py's *extras_require* (see example under [Creating new data types](#creating-new-data-types)). For example:

```python from setuptools import setup

**setup(** ..., entry_points={

**"pytest_wdl.url_schemes": [** "myexec = pytest_wdl.url_schemes.myscheme:MyUrlScheme"

]

}, extras_require={

"myexec": ["mylib"]

}
```

CHAPTER

# ELEVEN

)

CHAPTER

TWELVE

# PYTEST_WDL

## 12.1 pytest_wdl package

### 12.1.1 Subpackages

#### pytest_wdl.data_types package

#### Submodules

#### pytest_wdl.data_types.bam module

Convert BAM to SAM for diff.

**class** pytest_wdl.data_types.bam.**BamDataFile**(*local_path: pathlib.Path, localizer: Optional[pytest_wdl.localizers.Localizer] = None, **compare_opts*)

Bases: *pytest_wdl.data_types.DataFile*

Supports comparing output of BAM file. This uses pysam to convert BAM to SAM, so that DataFile can carry out a regular diff on the SAM files.

**class** pytest_wdl.data_types.bam.**Sorting**(*value*)

Bases: enum.Enum

An enumeration.

**COORDINATE = 1**

**NAME = 2**

**NONE = 0**

pytest_wdl.data_types.bam.**assert_bam_files_equal**(*file1: pathlib.Path, file2: pathlib.Path, allowed_diff_lines: int = 0, min_mapq: int = 0, compare_tag_columns: bool = False*)

Compare two BAM files: * Convert them to SAM format * Optionally re-sort the files by chromosome, position, and flag * First compare all lines using only a subset of columns that should be deterministic * Next, filter the files by MAPQ and compare the remaining rows using all columns

**Parameters**

- **file1** – First BAM to compare
- **file2** – Second BAM to compare

- **allowed_diff_lines** – Number of lines by which the BAMs are allowed to differ (after being convert to SAM)
- **min_mapq** – Minimum mapq used to filter reads when comparing all columns
- **compare_tag_columns** – Whether to include tag columns (12+) when comparing all columns

pytest_wdl.data_types.bam.**bam_to_sam**(*input_bam: pathlib.Path, output_sam: pathlib.Path, headers: Optional[Iterable[str]] = ('HD', 'SQ', 'RG'), min_mapq: Optional[int] = None, sorting: pytest_wdl.data_types.bam.Sorting = <Sorting.NONE: 0>*)

Use PySAM to convert bam to sam.

pytest_wdl.data_types.bam.**diff_bam_columns**(*file1: pathlib.Path, file2: pathlib.Path, columns: str*) → int

### pytest_wdl.data_types.json module

**class** pytest_wdl.data_types.json.**JsonDataFile**(*local_path: pathlib.Path, localizer: Optional[pytest_wdl.localizers.Localizer] = None, **compare_opts*)

Bases: *pytest_wdl.data_types.DataFile*

### pytest_wdl.data_types.vcf module

Some tools that generate VCF (callers) will result in very slightly different qual scores and other floating-point-valued fields when run on different hardware. This handler ignores the QUAL and INFO columns and only compares the genotype (GT) field of sample columns. Only works for single-sample VCFs.

**class** pytest_wdl.data_types.vcf.**VcfDataFile**(*local_path: pathlib.Path, localizer: Optional[pytest_wdl.localizers.Localizer] = None, **compare_opts*)

Bases: *pytest_wdl.data_types.DataFile*

pytest_wdl.data_types.vcf.**diff_vcf_columns**(*file1: pathlib.Path, file2: pathlib.Path, compare_phase: bool = False*) → int

### Module contents

**class** pytest_wdl.data_types.**DataFile**(*local_path: pathlib.Path, localizer: Optional[pytest_wdl.localizers.Localizer] = None, **compare_opts*)

Bases: `object`

A data file, which may be local, remote, or represented as a string.

**Parameters**

- **local_path** – Path where the data file should exist after being localized.
- **localizer** – Localizer object, for persisting the file on the local disk.
- **allowed_diff_lines** – Number of lines by which the file is allowed to differ from another and still be considered equal.
- **compare_opts** – Additional type-specific comparison options.

**assert_contents_equal**(*other: Union[str, pathlib.Path,* DataFile*]*) → None

Assert the contents of two files are equal.

If *allowed_diff_lines == 0*, files are compared using MD5 hashes, otherwise their contents are compared using the linux *diff* command.

**Parameters** **other** – A *DataFile* or string file path.

**Raises** **AssertionError if the files are different.** –

**property path**

**set_compare_opts**(***kwargs*)

Update comparison options.

**Parameters** ****kwargs** – Comparison options to update.

**class** pytest_wdl.data_types.**DefaultDataFile**(*local_path: pathlib.Path, localizer: Optional[*pytest_wdl.localizers.Localizer*] = None, **compare_opts*)

Bases: *pytest_wdl.data_types.DataFile*

pytest_wdl.data_types.**assert_binary_files_equal**(*file1: pathlib.Path, file2: pathlib.Path, digest: str = 'md5'*) → None

pytest_wdl.data_types.**assert_text_files_equal**(*file1: pathlib.Path, file2: pathlib.Path, allowed_diff_lines: int = 0, diff_fn: Callable[[pathlib.Path, pathlib.Path], int] = <function diff_default>*) → None

pytest_wdl.data_types.**compare_gzip**(*file1: pathlib.Path, file2: pathlib.Path*)

pytest_wdl.data_types.**diff_default**(*file1: pathlib.Path, file2: pathlib.Path*) → int

Default diff command.

**Parameters**

- **file1** – First file to compare
- **file2** – Second file to compare

**Returns** Number of different lines.

## pytest_wdl.executors package

### Submodules

### pytest_wdl.executors.cromwell_local module

**class** pytest_wdl.executors.cromwell_local.**CromwellLocalExecutor**(*import_dirs: Optional[Sequence[pathlib.Path]] = None, java_bin: Optional[Union[str, pathlib.Path]] = None, java_args: Optional[str] = None, cromwell_jar_file: Optional[Union[str, pathlib.Path]] = None, cromwell_configuration: Optional[Union[str, pathlib.Path, dict]] = None, cromwell_args: Optional[str] = None, cromwell_config_file: Optional[Union[str, pathlib.Path]] = None*)

Bases: *pytest_wdl.executors.JavaExecutor*, pytest_wdl.executors._cromwell.CromwellHelperMixin

Manages the running of WDL workflows using Cromwell.

**Parameters**

- **import_dirs** – Relative or absolute paths to directories containing WDL scripts that should be available as imports.
- **java_bin** – Path to the java executable.
- **java_args** – Default Java arguments to use; can be overidden by passing *java_args=...* to *run_workflow*.
- **cromwell_jar_file** – Path to the Cromwell JAR file.
- **cromwell_args** – Default Cromwell arguments to use; can be overridden by passing *cromwell_args=...* to *run_workflow*.

**run_workflow**(*wdl_path: pathlib.Path, inputs: Optional[dict] = None, expected: Optional[dict] = None, **kwargs*) → dict

Run a WDL workflow on given inputs, and check that the output matches given expected values.

**Parameters**

- **wdl_path** – The WDL script to execute.
- **inputs** – Object that will be serialized to JSON and provided to Cromwell as the workflow inputs.
- **expected** – Dict mapping output parameter names to expected values.
- **kwargs** – Additional keyword arguments, mostly for debugging: * workflow_name: The name of the workflow in the WDL script. If None,

  the name of the WDL script is used (without the .wdl extension).

  - **inputs_file: Path to the Cromwell inputs file to use. Inputs are** written to this file only if it doesn't exist.
  - **imports_file: Path to the WDL imports file to use. Imports are** written to this file only if it doesn't exist.
  - java_args: Additional arguments to pass to Java runtime.
  - cromwell_args: Additional arguments to pass to *cromwell run*.

**Returns** Dict of outputs.

**Raises**

- ***ExecutionFailedError*** – if there was an error executing Cromwell
- **AssertionError** – if the actual outputs don't match the expected outputs

### pytest_wdl.executors.cromwell_server module

**class** pytest_wdl.executors.cromwell_server.**CromwellServerExecutor**(*import_dirs: Optional[Sequence[pathlib.Path]] = None, cromwell_api_url: Optional[str] = 'http://localhost:8000/api/workflows/ cromwell_api_username: Optional[str] = None, cromwell_api_password: Optional[str] = None, cromwell_configuration: Optional[Union[str, pathlib.Path, dict]] = None*)

Bases: *pytest_wdl.executors.Executor*, pytest_wdl.executors._cromwell.

`CromwellHelperMixin`

Manages the running of WDL workflows using a remote Cromwell running in Server mode.

**Parameters**

- **`import_dirs`** – Relative or absolute paths to directories containing WDL scripts that should be available as imports.
- **`cromwell_api_url`** – The full URL where this cromwell exists *http://localhost:8000/api/workflows/v1*
- **`cromwell_api_username`** – The username to pass to the cromwell API if protected by basic auth
- **`cromwell_api_password`** – The password to pass to the cromwell API if protected by basic auth
- **`cromwell_configuration`** – A config file that will be passed to Cromwell

**`run_workflow`**(*wdl_path: pathlib.Path*, *inputs: Optional[dict] = None*, *expected: Optional[dict] = None*, ***kwargs*) → dict

Run a WDL workflow on given inputs, and check that the output matches given expected values.

**Parameters**

- **`wdl_path`** – The WDL script to execute.
- **`inputs`** – Object that will be serialized to JSON and provided to Cromwell as the workflow inputs.
- **`expected`** – Dict mapping output parameter names to expected values.
- **`kwargs`** – Additional keyword arguments, mostly for debugging: * workflow_name: The name of the workflow in the WDL script. If None,

  the name of the WDL script is used (without the .wdl extension).

  - **inputs_file: Path to the Cromwell inputs file to use. Inputs are** written to this file only if it doesn't exist.
  - **imports_file: Path to the WDL imports file to use. Imports are** written to this file only if it doesn't exist.
  - java_args: Additional arguments to pass to Java runtime.
  - cromwell_args: Additional arguments to pass to *cromwell run*.

**Returns** Dict of outputs.

**Raises**

- ***`ExecutionFailedError`*** – if there was an error executing Cromwell
- **`AssertionError`** – if the actual outputs don't match the expected outputs

### pytest_wdl.executors.miniwdl module

**class** pytest_wdl.executors.miniwdl.**MiniwdlExecutor**(*import_dirs: Optional[Sequence[pathlib.Path]] = None*)

Bases: *pytest_wdl.executors.Executor*

Manages the running of WDL workflows using Cromwell.

**static log_source**(*logger: logging.Logger*, *exn: Exception*)

**static read_miniwdl_command_std**(*path: Optional[str] = None*) → Optional[str]

**run_workflow**(*wdl_path: pathlib.Path*, *inputs: Optional[dict] = None*, *expected: Optional[dict] = None*, ***kwargs*) → dict

Run a WDL workflow on given inputs, and check that the output matches given expected values.

**Parameters**

- **wdl_path** – The WDL script to execute.
- **inputs** – Object that will be serialized to JSON and provided to Cromwell as the workflow inputs.
- **expected** – Dict mapping output parameter names to expected values.
- **kwargs** – Additional keyword arguments, mostly for debugging: * workflow_name: Name of the workflow to run. * task_name: Name of the task to run if a workflow isn't defined. * inputs_file: Path to the Cromwell inputs file to use. Inputs are

  written to this file only if it doesn't exist.

**Returns** Dict of outputs.

**Raises**

- **Exception** – if there was an error executing Cromwell
- **AssertionError** – if the actual outputs don't match the expected outputs

### Module contents

**exception** pytest_wdl.executors.**ExecutionFailedError**(*executor: str*, *target: str*, *status: str*, *inputs: Optional[dict] = None*, *executor_stdout: Optional[str] = None*, *executor_stderr: Optional[str] = None*, *failed_task: Optional[str] = None*, *failed_task_exit_status: Optional[int] = None*, *failed_task_stdout: Optional[str] = None*, *failed_task_stderr: Optional[str] = None*, *msg: Optional[str] = None*)

Bases: *pytest_wdl.executors.ExecutorError*

**property exit_status_str**

**class** pytest_wdl.executors.**Executor**

Bases: object

Base class for WDL workflow executors.

**abstract run_workflow**(*wdl_path: pathlib.Path, inputs: Optional[dict] = None, expected: Optional[dict] = None, **kwargs*) → dict

Run a WDL workflow on given inputs, and check that the output matches given expected values.

**Parameters**

- **wdl_path** – The WDL script to execute.
- **inputs** – Object that will be serialized to JSON and provided to Cromwell as the workflow inputs.
- **expected** – Dict mapping output parameter names to expected values.
- **kwargs** – Additional executor-specific keyword arguments (mostly for debugging)

**Returns** Dict of outputs.

**Raises**

- ***ExecutionFailedError*** – if there was an error executing the workflow
- **AssertionError** – if the actual outputs don't match the expected outputs

**exception** pytest_wdl.executors.**ExecutorError**(*executor: str, msg: Optional[str] = None*)

Bases: Exception

**class** pytest_wdl.executors.**InputsFormatter**

Bases: object

**format_inputs**(*inputs_dict: dict, namespace: Optional[str] = None*) → dict

**format_value**(*value: Any*) → Any

Convert a primitive, DataFile, Sequence, or Dict to a JSON-serializable object. Currently, arbitrary objects can be serialized by implementing an *as_dict()* method, otherwise they are converted to strings.

**Parameters** **value** – The value to format.

**Returns** The serializable value.

**classmethod get_instance**() → *pytest_wdl.executors.InputsFormatter*

**class** pytest_wdl.executors.**JavaExecutor**(*java_bin: Optional[Union[str, pathlib.Path]] = None, java_args: Optional[str] = None*)

Bases: *pytest_wdl.executors.Executor*

Manages the running of WDL workflows using a Java-based executor.

**Parameters**

- **java_bin** – Path to the java executable.
- **java_args** – Default Java arguments to use; can be overidden by passing *java_args=...* to *run_workflow*.

**static resolve_jar_file**(*file_name_pattern: str, jar_path: Optional[pathlib.Path] = None, env_var: Optional[str] = None*)

pytest_wdl.executors.**get_target_name**(*wdl_path: Optional[pathlib.Path] = None, wdl_doc: Optional[WDL.Tree.Document] = None, task_name: Optional[str] = None, workflow_name: Optional[str] = None, **kwargs*) → Tuple[str, bool]

Get the execution target. The order of priority is:

- task_name
- workflow_name

- wdl_doc.workflow.name
- wdl_doc.task[0].name
- wdl_file.stem

**Parameters**

- **wdl_path** – Path to a WDL file
- **wdl_doc** – A miniwdl-parsed WDL document
- **task_name** – The task name
- **workflow_name** – The workflow name
- ****kwargs** – Additional keyword arguments to pass to *parse_wdl*

**Returns** A tuple (target, is_task), where *is_task* is a boolean indicating whether the target is a task (True) or a workflow (False).

**Raises**

- **ValueError if 1) neither task_name nor workflow_name is specified and the** –
- **WDL document contains no workflow and multiple tasks; or 2) all of the** –
- **parameters are None.** –

`pytest_wdl.executors.`**`parse_wdl`**(*wdl_path: pathlib.Path, import_dirs: Optional[Sequence[pathlib.Path]] = (), check_quant: bool = False, **_*) → WDL.Tree.Document

`pytest_wdl.executors.`**`read_write_inputs`**(*inputs_file: Optional[Union[str, pathlib.Path]] = None, inputs_dict: Optional[dict] = None, inputs_formatter: Optional[pytest_wdl.executors.InputsFormatter] = <pytest_wdl.executors.InputsFormatter object>, write_formatted_inputs: bool = True, **kwargs*) → Tuple[dict, Optional[pathlib.Path]]

If *inputs_file* is specified and it exists, read its contents. Otherwise, if *inputs_dict* is specified, format it using *inputs_formatter* (if specified) and write it to *inputs_file* or a temporary file.

**Parameters**

- **inputs_file** –
- **inputs_dict** –
- **inputs_formatter** –
- **write_formatted_inputs** –
- **kwargs** –

**Returns** The (formatted) inputs dict and the resolved inputs file. If both *inputs_dict* and *inputs_file* are None, returns *({}, None)*.

## pytest_wdl.providers package

### Submodules

### pytest_wdl.providers.dx module

**class** pytest_wdl.providers.dx.**DxInputsFormatter**(*wdl_doc: WDL.Tree.Document, project_id: str = None, data_project_id: Optional[str] = None, folder: str = '/', data_folder: Optional[str] = None, **_*)

Bases: object

**format_inputs**(*inputs_dict: dict, namespace: Optional[str] = None*) → dict

**format_value**(*value: Any, path: Tuple[str, ...], nested: bool = False*) → Tuple[Any, bool]

Convert a primitive, DataFile, Sequence, or Dict to a JSON-serializable object. Currently, arbitrary objects can be serialized by implementing an *as_dict()* method, otherwise they are converted to strings.

**Parameters**

- **value** – The value to format.
- **path** – The path to the current value.
- **nested** – Whether the value is nested within a complex type

**Returns** The tuple *(val, is_complex)*, where *val* is serializable value and *is_complex* is True if the value is a complex type.

**class** pytest_wdl.providers.dx.**DxResponse**(*file_id: str, project_id: Optional[str] = None*)

Bases: *pytest_wdl.url_schemes.Response*

**download_file**(*destination: pathlib.Path, show_progress: bool = False, digests: Optional[dict] = None*)

Download a file to a specific destination.

**Parameters**

- **destination** – Destination path
- **show_progress** – Whether to show a progress bar
- **digests** – Optional dict mapping hash names to digests. These are used to validate the downloaded file.

**Raises** ***DigestsNotEqualError*** –

**class** pytest_wdl.providers.dx.**DxUrlHandler**

Bases: *pytest_wdl.url_schemes.UrlHandler*

**property handles**

**property scheme**

**urlopen**(*request: urllib.request.Request*) → *pytest_wdl.url_schemes.Response*

**class** pytest_wdl.providers.dx.**DxWdlExecutor**(*import_dirs: Optional[Sequence[pathlib.Path]] = None, java_bin: Optional[Union[str, pathlib.Path]] = None, java_args: Optional[str] = None, dxwdl_jar_file: Optional[Union[str, pathlib.Path]] = None, dxwdl_cache_dir: Optional[Union[str, pathlib.Path]] = None*)

Bases: *pytest_wdl.executors.JavaExecutor*

**run_workflow**(*wdl_path: pathlib.Path, inputs: Optional[dict] = None, expected: Optional[dict] = None, **kwargs*) → dict

Run a WDL workflow on given inputs, and check that the output matches given expected values.

**Parameters**

- **wdl_path** – The WDL script to execute.
- **inputs** – Object that will be serialized to JSON and provided to Cromwell as the workflow inputs.
- **expected** – Dict mapping output parameter names to expected values.
- **kwargs** – Additional executor-specific keyword arguments (mostly for debugging)

**Returns** Dict of outputs.

**Raises**

- ***ExecutionFailedError*** – if there was an error executing the workflow
- **AssertionError** – if the actual outputs don't match the expected outputs

pytest_wdl.providers.dx.**login**(*logout: bool = False, interactive: bool = True*)

Checks that the user is logged into DNAnexus, otherwise log them in.

**Parameters**

- **logout** – Whether to log out before exiting the context. Ignored if the user is already logged in.
- **interactive** – Whether to allow interactive login.

**Raises** ***PluginError*** – if *interactive* is *False* and the user cannot be logged in non-interactively

#### Module contents

### 12.1.2 Submodules

### 12.1.3 pytest_wdl.config module

**class** pytest_wdl.config.**UserConfiguration**(*config_file: Optional[pathlib.Path] = None, cache_dir: Optional[pathlib.Path] = None, remove_cache_dir: Optional[bool] = None, execution_dir: Optional[pathlib.Path] = None, proxies: Optional[Dict[str, Union[str, Dict[str, str]]]] = None, http_headers: Optional[List[dict]] = None, show_progress: Optional[bool] = None, executors: Optional[str] = None, executor_defaults: Optional[Dict[str, dict]] = None, provider_defaults: Optional[Dict[str, dict]] = None*)

Bases: `object`

Stores pytest-wdl configuration. If configuration options are specified both in the config file and as arguments to the constructor, the latter take precedence.

**Parameters**

- **config_file** – JSON (or YAML) file from which to load default values.
- **cache_dir** – The directory in which to cache localized files; defaults to using a temporary directory that is specific to each module and deleted afterwards.
- **remove_cache_dir** – Whether to remove the cache directory; if None, takes the value True if a temp directory is used for caching, and False, if a value for *cache_dir* is specified.
- **execution_dir** – The directory in which to run workflows. Defaults to None, which signals that a different temporary directory should be used for each workflow run.
- **proxies** – Mapping of proxy type (typically 'http' or 'https' to either an environment variable, or a dict with either/both keys 'env' and 'value', where the value is taken from the environment variable ('env') first, and from 'value' if the environment variable is not specified or is unset.
- **http_headers** – A list of dicts, each of which defines a header. The allowed keys are 'pattern', 'name', 'env', and 'value', where pattern is a URL pattern to match, 'name' is the header name and 'env' and 'value' are interpreted the same as for *proxies*. If no pattern is provided, the header is used for all URLs.
- **show_progress** – Whether to show progress bars when downloading remote test data files.
- **executors** – Default set of executors to run.
- **executor_defaults** – Mapping of executor name to dict of executor-specific configuration options.

**as_dict**() → dict

**cleanup**() → None

Preforms cleanup operations, such as deleting the cache directory if *self.remove_cache_dir* is True.

**get_executor_defaults**(*executor_name: str*) → dict
Get default configuration values for the given executor.

**Parameters** **executor_name** – The executor name

**Returns** A dict with the executor configuration values, if any.

**get_provider_defaults**(*provider_name: str*) → dict
Get default configuration values for the given provider.

**Parameters** **provider_name** – The provider name

**Returns** A dict with the provider configuration values, if any.

**save**(*path: pathlib.Path*) → None

pytest_wdl.config.**cleanup**()

pytest_wdl.config.**default_user_config_file**() → pathlib.Path

pytest_wdl.config.**get_instance**() → *pytest_wdl.config.UserConfiguration*

pytest_wdl.config.**set_instance**(*config: Optional[*pytest_wdl.config.UserConfiguration*] = None*, *path: Optional[pathlib.Path] = None*)

### 12.1.4 pytest_wdl.core module

pytest_wdl.core.**DATA_TYPES = {'bam': <pytest_wdl.plugins.PluginFactory object>, 'json': <**
Data type plugin modules from the discovered entry points.

**class** pytest_wdl.core.**DataDirs**(*basedir: pathlib.Path*, *module: Optional[Union[str, Any]] = None*, *function: Optional[Union[str, Callable]] = None*, *cls: Optional[Union[str, Type]] = None*)

Bases: object

Provides data files from test data directory structure as defined by the datadir and datadir-ng plugins. Paths are resolved lazily upon first request.

**property paths**

**class** pytest_wdl.core.**DataManager**(*data_resolver:* pytest_wdl.core.DataResolver, *datadirs:* pytest_wdl.core.DataDirs)

Bases: object

Manages test data, which is defined in a test_data.json file.

**Parameters**

- **data_resolver** – Module-level config.
- **datadirs** – Data directories to search for the data file.

**get_dict**(**names: str*, ***params*) → dict
Creates a dict with one or more entries from this DataManager.

**Parameters**

- ***names** – Names of test data entries to add to the dict.
- ****params** – Mapping of workflow parameter names to test data entry names.

**Returns** Dict mapping parameter names to test data entries for all specified names.

**get_list**(**names: str*) → list

**class** pytest_wdl.core.**DataResolver**(*data_descriptors: dict, user_config: pytest_wdl.config.UserConfiguration*)

Bases: object

Resolves data files that may need to be localized.

**resolve**(*name: str, datadirs: Optional[pytest_wdl.core.DataDirs] = None*)

pytest_wdl.core.**EXECUTORS = {'cromwell': <pytest_wdl.plugins.PluginFactory object>, 'cromw**

Executor plugin modules from the discovered entry points.

pytest_wdl.core.**create_data_file**(*user_config: pytest_wdl.config.UserConfiguration, type: Optional[Union[str, dict]] = 'default', name: Optional[str] = None, path: Optional[Union[str, pathlib.Path]] = None, url: Optional[str] = None, contents: Optional[Union[str, dict]] = None, env: Optional[str] = None, http_headers: Optional[dict] = None, digests: Optional[dict] = None, datadirs: Optional[pytest_wdl.core.DataDirs] = None, **kwargs*) → *pytest_wdl.data_types.DataFile*

pytest_wdl.core.**create_executor**(*executor_name: str, import_dirs: Sequence[pathlib.Path], user_config: pytest_wdl.config.UserConfiguration*)

### 12.1.5 pytest_wdl.fixtures module

Fixtures for writing tests that execute WDL workflows using Cromwell.

Note: This library is being transitioned to python3 only, and to use `pathlib.Path`s instead of string paths. For backward compatibility fixtures that produce a path may still return string paths, but this support will be dropped in a future version.

**class** pytest_wdl.fixtures.**WorkflowRunner**(*wdl_search_paths: Sequence[pathlib.Path], import_dirs: Sequence[pathlib.Path], user_config: pytest_wdl.config.UserConfiguration, subtests: pytest_subtests.SubTests, default_executors: Optional[Sequence[str]] = None*)

Bases: object

Callable object that runs tests.

pytest_wdl.fixtures.**default_executors**(*user_config: pytest_wdl.config.UserConfiguration*) → Sequence[str]

pytest_wdl.fixtures.**import_dirs**(*request: _pytest.fixtures.FixtureRequest, project_root: Union[str, pathlib.Path], import_paths: Optional[Union[str, pathlib.Path]]*) → List[Union[str, pathlib.Path]]

Fixture that provides a list of directories containing WDL scripts to make avaialble as imports. Uses the file provided by *import_paths* fixture if it is not None, otherwise returns a list containing the parent directory of the test module.

**Parameters**

- **request** – A FixtureRequest object
- **project_root** – Project root directory
- **import_paths** – File listing paths to imports, one per line

pytest_wdl.fixtures.**import_paths**(*request: _pytest.fixtures.FixtureRequest*) → Optional[Union[str, pathlib.Path]]

Fixture that provides the path to a file that lists directories containing WDL scripts to make available as imports. This looks for the file at "tests/import_paths.txt" by default, and returns None if that file doesn't exist.

`pytest_wdl.fixtures.`**`project_root`**(*request: _pytest.fixtures.FixtureRequest, project_root_files: List[str]*) → Union[str, pathlib.Path]

Fixture that provides the root directory of the project. By default, this assumes that the project has one subdirectory per task, and that this framework is being run from the test subdirectory of a task diretory, and therefore looks for the project root two directories up.

`pytest_wdl.fixtures.`**`project_root_files`**() → List[str]

Fixture that provides a list of filenames that are found in the project root directory. Used by the *project_root* fixture to locate the project root directory.

`pytest_wdl.fixtures.`**`user_config`**(*user_config_file: Optional[pathlib.Path]*) → *pytest_wdl.config.UserConfiguration*

`pytest_wdl.fixtures.`**`user_config_file`**() → Optional[pathlib.Path]

Fixture that provides the value of 'user_config' environment variable. If not specified, looks in the default location ($HOME/pytest_user_config.json).

**Returns** Path to the confif file, or None if not specified.

`pytest_wdl.fixtures.`**`workflow_data`**(*request: _pytest.fixtures.FixtureRequest, workflow_data_resolver:* pytest_wdl.core.DataResolver) → *pytest_wdl.core.DataManager*

Provides an accessor for test data files, which may be local or in a remote repository.

**Parameters**

- **`request`** – FixtureRequest object
- **`workflow_data_resolver`** – Module-level test data configuration

### Examples

**def workflow_data_descriptor_file():** return "tests/test_data.json"

**def test_workflow(workflow_data):** print(workflow_data["myfile"])

`pytest_wdl.fixtures.`**`workflow_data_descriptor_file`**(*request: _pytest.fixtures.FixtureRequest*) → Union[str, pathlib.Path]

Fixture that provides the path to the JSON file that describes test data files.

**Parameters** **`request`** – A FixtureRequest object

`pytest_wdl.fixtures.`**`workflow_data_descriptors`**(*request: _pytest.fixtures.FixtureRequest, project_root: Union[str, pathlib.Path], workflow_data_descriptor_file: Union[str, pathlib.Path]*) → dict

Fixture that provides a mapping of test data names to values. If workflow_data_descriptor_file is relative, it is searched first relative to the current test context directory and then relative to the project root.

**Parameters** **`workflow_data_descriptor_file`** – Path to the data descriptor JSON file.

**Returns** A dict with keys as test data names and each value either a primitive, a map describing a data file, or a DataFile object.

`pytest_wdl.fixtures.`**`workflow_data_resolver`**(*workflow_data_descriptors: dict, user_config:* pytest_wdl.config.UserConfiguration) → *pytest_wdl.core.DataResolver*

Provides access to test data files for tests in a module.

**Parameters**

- **workflow_data_descriptors** – workflow_data_descriptors fixture.
- **user_config** –

pytest_wdl.fixtures.**workflow_runner**(*request: _pytest.fixtures.FixtureRequest, project_root: Union[str, pathlib.Path], import_dirs: List[Union[str, pathlib.Path]], user_config:* pytest_wdl.config.UserConfiguration, *default_executors: Sequence[str], subtests: pytest_subtests.SubTests*)

Provides a callable that runs a workflow. The callable has the same signature as *Executor.run_workflow*, but takes an additional keyword argument *executors*, a sequence of strings, which allows overriding the names of the executors to use.

If multiple executors are specified, the tests are run using the *subtests* fixture of the *pytest-subtests* plugin.

**Parameters**

- **request** – A FixtureRequest object.
- **project_root** – Project root directory.
- **import_dirs** – Directories from which to import WDL scripts.
- **user_config** – A UserConfiguration object.
- **default_executors** – Names of executors to use when executor name isn't passed to the *workflow_runner* callable.
- **subtests** – A SubTests object.

**Returns** A generator over the results of calling the workflow with each executor. Each value is a tuple *(executor_name, execution_dir, outputs)*, where *execution_dir* is the root directory where the task/workflow was run (the structure of the directory is executor-dependent) and *outputs* is a dict of the task/workflow outputs.

### 12.1.6 pytest_wdl.loader module

**class** pytest_wdl.loader.**JsonWdlTestsModule**(*fspath: py._path.local.LocalPath, parent=None, config=None, session=None, nodeid=None*)

Bases: *pytest_wdl.loader.WdlTestsModule*

**class** pytest_wdl.loader.**TestItem**(*parent, data: Optional[dict] = None, name: Optional[str] = None, wdl: Optional[str] = None, inputs: Optional[dict] = None, expected: Optional[dict] = None, tags: Optional[Sequence] = None, **kwargs*)

Bases: _pytest.nodes.Item

**runtest**()

**setup**()

This method is black magic - uses internal pytest APIs to create a FixtureRequest that can be used to access fixtures in *runtest()*. Copied from https://github.com/pytest-dev/pytest/blob/master/src/_pytest/doctest.py.

**class** pytest_wdl.loader.**WdlTestsModule**(*fspath: py._path.local.LocalPath, parent=None, config=None, session=None, nodeid=None*)

Bases: _pytest.python.Module

**collect**()

returns a list of children (items and collectors) for this collection node.

**class** pytest_wdl.loader.**YamlWdlTestsModule**(*fspath: py._path.local.LocalPath, parent=None, config=None, session=None, nodeid=None*)

Bases: *pytest_wdl.loader.WdlTestsModule*

pytest_wdl.loader.**pytest_collect_file**(*path: py._path.local.LocalPath, parent*) → Optional[_pytest.nodes.File]

pytest_wdl.loader.**pytest_collection**(*session: _pytest.main.Session*)

Prints an empty line to make the report look slightly better.

### 12.1.7 pytest_wdl.localizers module

**class** pytest_wdl.localizers.**JsonLocalizer**(*contents: dict*)

Bases: *pytest_wdl.localizers.Localizer*

**localize**(*destination: pathlib.Path*)

Localize a resource to *destination*.

**Parameters** **destination** – Path to file where the non-local resource is to be localized.

**class** pytest_wdl.localizers.**LinkLocalizer**(*source: pathlib.Path*)

Bases: *pytest_wdl.localizers.Localizer*

Localizes a file to another destination using a symlink.

**localize**(*destination: pathlib.Path*)

Localize a resource to *destination*.

**Parameters** **destination** – Path to file where the non-local resource is to be localized.

**class** pytest_wdl.localizers.**Localizer**

Bases: object

Abstract base of classes that implement file localization.

**abstract localize**(*destination: pathlib.Path*) → None

Localize a resource to *destination*.

**Parameters** **destination** – Path to file where the non-local resource is to be localized.

**verify**(*path: pathlib.Path*) → bool

Verify that *path* exists and is valid.

**Parameters** **path** – Path to verify.

**Returns** True if the path is verified, else False

**class** pytest_wdl.localizers.**StringLocalizer**(*contents: str*)

Bases: *pytest_wdl.localizers.Localizer*

Localizes a string by writing it to a file.

**localize**(*destination: pathlib.Path*)

Localize a resource to *destination*.

**Parameters** **destination** – Path to file where the non-local resource is to be localized.

**class** pytest_wdl.localizers.**UrlLocalizer**(*url: str, user_config: pytest_wdl.config.UserConfiguration, http_headers: Optional[dict] = None, digests: Optional[dict] = None*)

Bases: *pytest_wdl.localizers.Localizer*

Localizes a file specified by a URL.

**property http_headers**

**localize**(*destination: pathlib.Path*)
Localize a resource to *destination*.

**Parameters** **destination** – Path to file where the non-local resource is to be localized.

**property proxies**

**verify**(*path: pathlib.Path*) → bool
Verify that *path* exists and is valid.

**Parameters** **path** – Path to verify.

**Returns** True if the path is verified, else False

pytest_wdl.localizers.**download_file**(*url: str, destination: pathlib.Path, http_headers: Optional[dict] = None, proxies: Optional[dict] = None, show_progress: bool = True, digests: Optional[dict] = None*)

### 12.1.8 pytest_wdl.plugins module

**exception** pytest_wdl.plugins.**PluginError**
Bases: Exception

**class** pytest_wdl.plugins.**PluginFactory**(*entry_point: pkg_resources.EntryPoint, return_type: Type[T]*)
Bases: typing.Generic

Lazily loads a plugin class associated with a data type.

pytest_wdl.plugins.**plugin_factory_map**(*return_type: Type[T], group: Optional[str] = None, entry_points: Optional[Iterable[pkg_resources.EntryPoint]] = None*) → Dict[str, *pytest_wdl.plugins.PluginFactory*[T]]
Creates a mapping of entry point name to *PluginFactory* for all discovered entry points in the specified group.

**Parameters**

- **group** – Entry point group name
- **return_type** – Expected return type
- **entry_points** –

**Returns** Dict mapping entry point name to *PluginFactory* instances

### 12.1.9 pytest_wdl.url_schemes module

**class** pytest_wdl.url_schemes.**BaseResponse**
Bases: *pytest_wdl.url_schemes.Response*

**download_file**(*destination: pathlib.Path, show_progress: bool = False, digests: Optional[dict] = None*)
Download a file to a specific destination.

**Parameters**

- **destination** – Destination path

- **show_progress** – Whether to show a progress bar
- **digests** – Optional dict mapping hash names to digests. These are used to validate the downloaded file.

**Raises** *DigestsNotEqualError* –

**abstract get_content_length**() → Optional[int]

**abstract read**(*block_size: int*)

**class** pytest_wdl.url_schemes.**Method**(*value*)

Bases: enum.Enum

An enumeration.

**OPEN = ('urlopen', '{}_open')**

**REQUEST = ('request', '{}_request')**

**RESPONSE = ('response', '{}_response')**

**class** pytest_wdl.url_schemes.**Response**

Bases: object

**abstract download_file**(*destination: pathlib.Path, show_progress: bool = False, digests: Optional[dict] = None*)

Download a file to a specific destination.

**Parameters**

- **destination** – Destination path
- **show_progress** – Whether to show a progress bar
- **digests** – Optional dict mapping hash names to digests. These are used to validate the downloaded file.

**Raises** *DigestsNotEqualError* –

**class** pytest_wdl.url_schemes.**ResponseWrapper**(*rsp*)

Bases: *pytest_wdl.url_schemes.BaseResponse*

**get_content_length**() → Optional[int]

**read**(*block_size: int*) → bytes

**class** pytest_wdl.url_schemes.**UrlHandler**

Bases: urllib.request.BaseHandler

**alias**()

Add aliases that are required by urllib for handled methods.

**property handles**

**request**(*request: urllib.request.Request*) → urllib.request.Request

**response**(*request: urllib.request.Request, response:* pytest_wdl.url_schemes.Response) → *pytest_wdl.url_schemes.Response*

**abstract property scheme**

**urlopen**(*request: urllib.request.Request*) → *pytest_wdl.url_schemes.Response*

pytest_wdl.url_schemes.**install_schemes**()

### 12.1.10 pytest_wdl.utils module

**exception** pytest_wdl.utils.**DigestsNotEqualError**
Bases: AssertionError

**exception** pytest_wdl.utils.**MaxCallException**(*last=None*)
Bases: *pytest_wdl.utils.PollingException*

Exception raised if maximum number of iterations is exceeded

**exception** pytest_wdl.utils.**PollingException**(*last=None*)
Bases: Exception

Base exception that stores the last result seen.

**exception** pytest_wdl.utils.**TimeoutException**(*last=None*)
Bases: *pytest_wdl.utils.PollingException*

Exception raised if polling function times out

pytest_wdl.utils.**chdir**(*todir: pathlib.Path*)
Context manager that temporarily changes directories.

**Parameters** **todir** – The directory to change to.

pytest_wdl.utils.**compare_files_with_hash**(*file1: pathlib.Path, file2: pathlib.Path, hash_name: str = 'md5'*)

pytest_wdl.utils.**context_dir**(*path: Optional[pathlib.Path] = None, change_dir: bool = False, cleanup: Optional[bool] = None*) → pathlib.Path
Context manager that looks for a specific environment variable to specify a directory. If the environment variable is not set, a temporary directory is created and cleaned up upon return from the yield.

**Parameters**

- **path** – The environment variable to look for.
- **change_dir** – Whether to change to the directory.
- **cleanup** – Whether to delete the directory when exiting the context. If None, the directory is only deleted if a temporary directory is created.

**Yields** A directory path.

pytest_wdl.utils.**ensure_path**(*path: Union[str, py._path.local.LocalPath, pathlib.Path], search_paths: Optional[Sequence[pathlib.Path]] = None, canonicalize: bool = True, exists: Optional[bool] = None, is_file: Optional[bool] = None, executable: Optional[bool] = None, create: bool = False*) → pathlib.Path
Converts a string path or py.path.local.LocalPath to a pathlib.Path.

**Parameters**

- **path** – The path to convert.
- **search_paths** – Directories to search for *path* if it is not already absolute. If *exists* is True, looks for the first search path that contains the file, otherwise just uses the first search path.
- **canonicalize** – Whether to return the canonicalized version of the path - expand home directory shortcut (~), make absolute, and resolve symlinks.
- **exists** – If True, raise an exception if the path does not exist; if False, raise an exception if the path does exist.

- **is_file** – If True, raise an exception if the path is not a file; if False, raise an exception if the path is not a directory.
- **executable** – If True and *is_file* is True and the file exists, raise an exception if it is not executable.
- **create** – Create the directory (or parent, if *is_file* = True) if it does not exist. Ignored if *exists* is True.

**Returns** A *pathlib.Path* object.

`pytest_wdl.utils.`**`env_map`**(*d: dict*) → dict

Given a mapping of keys to value descriptors, creates a mapping of the keys to the described values.

`pytest_wdl.utils.`**`find_executable_path`**(*executable: str*, *search_path: Optional[Sequence[pathlib.Path]] = None*) → Optional[pathlib.Path]

Finds 'executable' in *search_path*.

**Parameters**

- **executable** – The name of the executable to find.
- **search_path** – The list of directories to search. If None, the system search path (defined by the $PATH environment variable) is used.

**Returns** Absolute path of the executable, or None if no matching executable was found.

`pytest_wdl.utils.`**`find_in_classpath`**(*glob: str*) → Optional[pathlib.Path]

Attempts to find a .jar file matching the specified glob pattern in the Java classpath.

**Parameters** **glob** – JAR filename pattern

**Returns** Path to the JAR file, or None if a matching file is not found.

`pytest_wdl.utils.`**`find_project_path`**(**filenames: Union[str, pathlib.Path]*, *start: Optional[pathlib.Path] = None*, *return_parent: bool = False*, *assert_exists: bool = False*) → Optional[pathlib.Path]

Starting from *path* folder and moving upwards, search for any of *filenames* and return the first path containing any one of them.

**Parameters**

- ***filenames** – Filenames to search. Either a string filename, or a sequence of string path elements.
- **start** – Starting folder
- **return_parent** – Whether to return the containing folder or the discovered file.
- **assert_exists** – Whether to raise an exception if a file cannot be found.

**Returns** A *Path*, or *None* if no folder is found that contains any of *filenames*. If *return_parent* is *False* and more than one of the files is found one of the files is randomly selected for return.

**Raises** **FileNotFoundError if the file cannot be found and assert_exists is True.** –

`pytest_wdl.utils.`**`hash_file`**(*path: pathlib.Path*, *hash_name: str = 'md5'*) → str

`pytest_wdl.utils.`**`is_executable`**(*path: pathlib.Path*) → bool

Checks if a path is executable.

**Parameters** **path** – The path to check

**Returns** True if *path* exists and is executable by the user, otherwise False.

`pytest_wdl.utils.poll`(*target: Callable, step: int = 1, args: Optional[Sequence] = None, kwargs: Optional[dict] = None, timeout: Optional[int] = None, max_tries: Optional[int] = None, check_success: Callable = <class 'bool'>, step_function: Optional[Callable[[int, int], int]] = None, ignore_exceptions: Sequence = ()*)

Poll by calling a target function until a certain condition is met. You must specify at least a target function to be called and the step – base wait time between each function call.

Vendored from the [polling](https://github.com/justiniso/polling) package.

**Parameters**

- **target** – The target callable
- **step** – Step defines the amount of time to wait (in seconds)
- **args** – Arguments to be passed to the target function
- **kwargs** – Keyword arguments to be passed to the target function
- **timeout** – The target function will be called until the time elapsed is greater than the maximum timeout (in seconds). NOTE that the actual execution time of the function *can* exceed the time specified in the timeout. For instance, if the target function takes 10 seconds to execute and the timeout is 21 seconds, the polling function will take a total of 30 seconds (two iterations of the target –20s which is less than the timeout–21s, and a final iteration)
- **max_tries** – Maximum number of times the target function will be called before failing
- **check_success** – A callback function that accepts the return value of the target function. It must return true if you want the polling function to stop and return this value. It must return false if you want to continue polling. You may also use this function to collect non-success values. The default is a callback that tests for truthiness (anything not False, 0, or empty collection).
- **step_function** – A callback function that accepts two arguments: current_step, num_tries; and returns the next step value. By default, this is constant, but you can also pass a function that will increase or decrease the step. As an example, you can increase the wait time between calling the target function by 10 seconds every iteration until the step is 100 seconds–at which point it should remain constant at 100 seconds

```
>>> def my_step_function(current_step: int, num_tries: int) ->␣
↪int:
>>>     return max(current_step + 10, 100)
```

- **ignore_exceptions** – You can specify a tuple of exceptions that should be caught and ignored on every iteration. If the target function raises one of these exceptions, it will be caught and the exception instance will be pushed to the queue of values collected during polling. Any other exceptions raised will be raised as normal.

**Returns** The first value from the target function that meets the condions of the check_success callback. By default, this will be the first value that is not None, 0, False, ‘’, or an empty collection.

`pytest_wdl.utils.resolve_file`(*filename: Union[str, pathlib.Path], project_root: pathlib.Path, assert_exists: bool = True*) → Optional[pathlib.Path]

Finds *filename* under *project_root* or in the project path.

**Parameters**

- **filename** – The filename, relative path, or absolute path to resolve.

- **project_root** – The project root dir.
- **assert_exists** – Whether to raise an error if the file cannot be found.

**Returns** A *pathlib.Path* object, or None if the file cannot be found and *assert_exists* is False.

**Raises** **FileNotFoundError if the file cannot be found and assert_exists is True.** –

pytest_wdl.utils.**resolve_value_descriptor**(*value_descriptor: Union[str, dict]*) → Optional

Resolves the value of a value descriptor, which may be an environment variable name, or a map with keys *env* (the environment variable name) and *value* (the value to use if *env* is not specified or if the environment variable is unset.

**Parameters** **value_descriptor** –

Returns:

pytest_wdl.utils.**safe_string**(*s: str*, *replacement: str* = '_') → str

Makes a string safe by replacing non-word characters.

**Parameters**

- **s** – The string to make safe
- **replacement** – The replacement stringj

**Returns** The safe string

pytest_wdl.utils.**tempdir**(*change_dir: bool = False*, *tmproot: Optional[pathlib.Path] = None*, *cleanup: Optional[bool] = True*) → pathlib.Path

Context manager that creates a temporary directory, yields it, and then deletes it after return from the yield.

**Parameters**

- **change_dir** – Whether to temporarily change to the temp dir.
- **tmproot** – Root directory in which to create temporary directories.
- **cleanup** – Whether to delete the temporary directory before exiting the context.

pytest_wdl.utils.**verify_digests**(*path: pathlib.Path*, *digests: dict*)

### 12.1.11 Module contents

Fixtures for writing tests that execute WDL workflows using Cromwell. For testability purposes, the implementaiton of these fixtures is done in the pytest_wdl.fixtures module.

CHAPTER

THIRTEEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

pytest_wdl, 55
pytest_wdl.config, 44
pytest_wdl.core, 45
pytest_wdl.data_types, 34
pytest_wdl.data_types.bam, 33
pytest_wdl.data_types.json, 34
pytest_wdl.data_types.vcf, 34
pytest_wdl.executors, 39
pytest_wdl.executors.cromwell_local, 36
pytest_wdl.executors.cromwell_server, 37
pytest_wdl.executors.miniwdl, 39
pytest_wdl.fixtures, 46
pytest_wdl.loader, 48
pytest_wdl.localizers, 49
pytest_wdl.plugins, 50
pytest_wdl.providers, 44
pytest_wdl.providers.dx, 42
pytest_wdl.url_schemes, 50
pytest_wdl.utils, 52

# INDEX


## A

alias() (*pytest_wdl.url_schemes.UrlHandler method*), 51
as_dict() (*pytest_wdl.config.UserConfiguration method*), 44
assert_bam_files_equal() (*in module pytest_wdl.data_types.bam*), 33
assert_binary_files_equal() (*in module pytest_wdl.data_types*), 35
assert_contents_equal() (*pytest_wdl.data_types.DataFile method*), 34
assert_text_files_equal() (*in module pytest_wdl.data_types*), 35

## B

bam_to_sam() (*in module pytest_wdl.data_types.bam*), 34
BamDataFile (*class in pytest_wdl.data_types.bam*), 33
BaseResponse (*class in pytest_wdl.url_schemes*), 50

## C

chdir() (*in module pytest_wdl.utils*), 52
cleanup() (*in module pytest_wdl.config*), 45
cleanup() (*pytest_wdl.config.UserConfiguration method*), 44
collect() (*pytest_wdl.loader.WdlTestsModule method*), 48
compare_files_with_hash() (*in module pytest_wdl.utils*), 52
compare_gzip() (*in module pytest_wdl.data_types*), 35
context_dir() (*in module pytest_wdl.utils*), 52
COORDINATE (*pytest_wdl.data_types.bam.Sorting attribute*), 33
create_data_file() (*in module pytest_wdl.core*), 46
create_executor() (*in module pytest_wdl.core*), 46
CromwellLocalExecutor (*class in pytest_wdl.executors.cromwell_local*), 36
CromwellServerExecutor (*class in pytest_wdl.executors.cromwell_server*), 37

## D

DATA_TYPES (*in module pytest_wdl.core*), 45
DataDirs (*class in pytest_wdl.core*), 45
DataFile (*class in pytest_wdl.data_types*), 34
DataManager (*class in pytest_wdl.core*), 45
DataResolver (*class in pytest_wdl.core*), 45
default_executors() (*in module pytest_wdl.fixtures*), 46
default_user_config_file() (*in module pytest_wdl.config*), 45
DefaultDataFile (*class in pytest_wdl.data_types*), 35
diff_bam_columns() (*in module pytest_wdl.data_types.bam*), 34
diff_default() (*in module pytest_wdl.data_types*), 35
diff_vcf_columns() (*in module pytest_wdl.data_types.vcf*), 34
DigestsNotEqualError, 52
download_file() (*in module pytest_wdl.localizers*), 50
download_file() (*pytest_wdl.providers.dx.DxResponse method*), 42
download_file() (*pytest_wdl.url_schemes.BaseResponse method*), 50
download_file() (*pytest_wdl.url_schemes.Response method*), 51
DxInputsFormatter (*class in pytest_wdl.providers.dx*), 42
DxResponse (*class in pytest_wdl.providers.dx*), 42
DxUrlHandler (*class in pytest_wdl.providers.dx*), 42
DxWdlExecutor (*class in pytest_wdl.providers.dx*), 42

## E

ensure_path() (*in module pytest_wdl.utils*), 52
env_map() (*in module pytest_wdl.utils*), 53
ExecutionFailedError, 39
Executor (*class in pytest_wdl.executors*), 39
ExecutorError, 40
EXECUTORS (*in module pytest_wdl.core*), 46
exit_status_str() (*pytest_wdl.executors.ExecutionFailedError*

property), 39

## F

find_executable_path() (in module pytest_wdl.utils), 53
find_in_classpath() (in module pytest_wdl.utils), 53
find_project_path() (in module pytest_wdl.utils), 53
format_inputs() (pytest_wdl.executors.InputsFormatter method), 40
format_inputs() (pytest_wdl.providers.dx.DxInputsFormatter method), 42
format_value() (pytest_wdl.executors.InputsFormatter method), 40
format_value() (pytest_wdl.providers.dx.DxInputsFormatter method), 42

## G

get_content_length() (pytest_wdl.url_schemes.BaseResponse method), 51
get_content_length() (pytest_wdl.url_schemes.ResponseWrapper method), 51
get_dict() (pytest_wdl.core.DataManager method), 45
get_executor_defaults() (pytest_wdl.config.UserConfiguration method), 44
get_instance() (in module pytest_wdl.config), 45
get_instance() (pytest_wdl.executors.InputsFormatter class method), 40
get_list() (pytest_wdl.core.DataManager method), 45
get_provider_defaults() (pytest_wdl.config.UserConfiguration method), 45
get_target_name() (in module pytest_wdl.executors), 40

## H

handles() (pytest_wdl.providers.dx.DxUrlHandler property), 42
handles() (pytest_wdl.url_schemes.UrlHandler property), 51
hash_file() (in module pytest_wdl.utils), 53
http_headers() (pytest_wdl.localizers.UrlLocalizer property), 50

## I

import_dirs() (in module pytest_wdl.fixtures), 46
import_paths() (in module pytest_wdl.fixtures), 46
InputsFormatter (class in pytest_wdl.executors), 40
install_schemes() (in module pytest_wdl.url_schemes), 51
is_executable() (in module pytest_wdl.utils), 53

## J

JavaExecutor (class in pytest_wdl.executors), 40
JsonDataFile (class in pytest_wdl.data_types.json), 34
JsonLocalizer (class in pytest_wdl.localizers), 49
JsonWdlTestsModule (class in pytest_wdl.loader), 48

## L

LinkLocalizer (class in pytest_wdl.localizers), 49
localize() (pytest_wdl.localizers.JsonLocalizer method), 49
localize() (pytest_wdl.localizers.LinkLocalizer method), 49
localize() (pytest_wdl.localizers.Localizer method), 49
localize() (pytest_wdl.localizers.StringLocalizer method), 49
localize() (pytest_wdl.localizers.UrlLocalizer method), 50
Localizer (class in pytest_wdl.localizers), 49
log_source() (pytest_wdl.executors.miniwdl.MiniwdlExecutor static method), 39
login() (in module pytest_wdl.providers.dx), 43

## M

MaxCallException, 52
Method (class in pytest_wdl.url_schemes), 51
MiniwdlExecutor (class in pytest_wdl.executors.miniwdl), 39
module
- pytest_wdl, 55
- pytest_wdl.config, 44
- pytest_wdl.core, 45
- pytest_wdl.data_types, 34
- pytest_wdl.data_types.bam, 33
- pytest_wdl.data_types.json, 34
- pytest_wdl.data_types.vcf, 34
- pytest_wdl.executors, 39
- pytest_wdl.executors.cromwell_local, 36
- pytest_wdl.executors.cromwell_server, 37
- pytest_wdl.executors.miniwdl, 39
- pytest_wdl.fixtures, 46
- pytest_wdl.loader, 48
- pytest_wdl.localizers, 49
- pytest_wdl.plugins, 50
- pytest_wdl.providers, 44

pytest_wdl.providers.dx, 42
pytest_wdl.url_schemes, 50
pytest_wdl.utils, 52

## N

NAME (*pytest_wdl.data_types.bam.Sorting attribute*), 33
NONE (*pytest_wdl.data_types.bam.Sorting attribute*), 33

## O

OPEN (*pytest_wdl.url_schemes.Method attribute*), 51

## P

parse_wdl() (*in module pytest_wdl.executors*), 41
path() (*pytest_wdl.data_types.DataFile property*), 35
paths() (*pytest_wdl.core.DataDirs property*), 45
plugin_factory_map() (*in module pytest_wdl.plugins*), 50
PluginError, 50
PluginFactory (*class in pytest_wdl.plugins*), 50
poll() (*in module pytest_wdl.utils*), 53
PollingException, 52
project_root() (*in module pytest_wdl.fixtures*), 46
project_root_files() (*in module pytest_wdl.fixtures*), 47
proxies() (*pytest_wdl.localizers.UrlLocalizer property*), 50
pytest_collect_file() (*in module pytest_wdl.loader*), 49
pytest_collection() (*in module pytest_wdl.loader*), 49
pytest_wdl
module, 55
pytest_wdl.config
module, 44
pytest_wdl.core
module, 45
pytest_wdl.data_types
module, 34
pytest_wdl.data_types.bam
module, 33
pytest_wdl.data_types.json
module, 34
pytest_wdl.data_types.vcf
module, 34
pytest_wdl.executors
module, 39
pytest_wdl.executors.cromwell_local
module, 36
pytest_wdl.executors.cromwell_server
module, 37
pytest_wdl.executors.miniwdl
module, 39
pytest_wdl.fixtures
module, 46
pytest_wdl.loader
module, 48
pytest_wdl.localizers
module, 49
pytest_wdl.plugins
module, 50
pytest_wdl.providers
module, 44
pytest_wdl.providers.dx
module, 42
pytest_wdl.url_schemes
module, 50
pytest_wdl.utils
module, 52

## R

read() (*pytest_wdl.url_schemes.BaseResponse method*), 51
read() (*pytest_wdl.url_schemes.ResponseWrapper method*), 51
read_miniwdl_command_std() (*pytest_wdl.executors.miniwdl.MiniwdlExecutor static method*), 39
read_write_inputs() (*in module pytest_wdl.executors*), 41
REQUEST (*pytest_wdl.url_schemes.Method attribute*), 51
request() (*pytest_wdl.url_schemes.UrlHandler method*), 51
resolve() (*pytest_wdl.core.DataResolver method*), 46
resolve_file() (*in module pytest_wdl.utils*), 54
resolve_jar_file() (*pytest_wdl.executors.JavaExecutor static method*), 40
resolve_value_descriptor() (*in module pytest_wdl.utils*), 55
Response (*class in pytest_wdl.url_schemes*), 51
RESPONSE (*pytest_wdl.url_schemes.Method attribute*), 51
response() (*pytest_wdl.url_schemes.UrlHandler method*), 51
ResponseWrapper (*class in pytest_wdl.url_schemes*), 51
run_workflow() (*pytest_wdl.executors.cromwell_local.CromwellLocal method*), 36
run_workflow() (*pytest_wdl.executors.cromwell_server.CromwellServe method*), 38
run_workflow() (*pytest_wdl.executors.Executor method*), 40
run_workflow() (*pytest_wdl.executors.miniwdl.MiniwdlExecutor method*), 39
run_workflow() (*pytest_wdl.providers.dx.DxWdlExecutor method*), 43
runtest() (*pytest_wdl.loader.TestItem method*), 48

## S


safe_string() (*in module pytest_wdl.utils*), 55
save() (*pytest_wdl.config.UserConfiguration method*), 45
scheme() (*pytest_wdl.providers.dx.DxUrlHandler property*), 42
scheme() (*pytest_wdl.url_schemes.UrlHandler property*), 51
set_compare_opts() (*pytest_wdl.data_types.DataFile method*), 35
set_instance() (*in module pytest_wdl.config*), 45
setup() (*pytest_wdl.loader.TestItem method*), 48
Sorting (*class in pytest_wdl.data_types.bam*), 33
StringLocalizer (*class in pytest_wdl.localizers*), 49


## T


tempdir() (*in module pytest_wdl.utils*), 55
TestItem (*class in pytest_wdl.loader*), 48
TimeoutException, 52


## U


UrlHandler (*class in pytest_wdl.url_schemes*), 51
UrlLocalizer (*class in pytest_wdl.localizers*), 49
urlopen() (*pytest_wdl.providers.dx.DxUrlHandler method*), 42
urlopen() (*pytest_wdl.url_schemes.UrlHandler method*), 51
user_config() (*in module pytest_wdl.fixtures*), 47
user_config_file() (*in module pytest_wdl.fixtures*), 47
UserConfiguration (*class in pytest_wdl.config*), 44


## V


VcfDataFile (*class in pytest_wdl.data_types.vcf*), 34
verify() (*pytest_wdl.localizers.Localizer method*), 49
verify() (*pytest_wdl.localizers.UrlLocalizer method*), 50
verify_digests() (*in module pytest_wdl.utils*), 55


## W


WdlTestsModule (*class in pytest_wdl.loader*), 48
workflow_data() (*in module pytest_wdl.fixtures*), 47
workflow_data_descriptor_file() (*in module pytest_wdl.fixtures*), 47
workflow_data_descriptors() (*in module pytest_wdl.fixtures*), 47
workflow_data_resolver() (*in module pytest_wdl.fixtures*), 47
workflow_runner() (*in module pytest_wdl.fixtures*), 48
WorkflowRunner (*class in pytest_wdl.fixtures*), 46


## Y


YamlWdlTestsModule (*class in pytest_wdl.loader*), 48