pytest-wdl

Release 1.1.1

John Didion, Michael T. Neylon

Sep 27, 2019

CONTENTS:

CHAPTER

ONE

USER MANUAL

pytest-wdl is a plugin for the pytest unit testing framework that enables testing of workflows written in Workflow Description Language. Test workflow inputs and expected outputs are *configured* in a `test_data.json` file. Workflows are run by one or more *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.

1.1 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
```

(continues on next page)

(continued from previous page)

```
|   |_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 is the first directory up in the directory hierarchy that contains a "tests" subdirectory. The test context directory may differ between test modules, depending on the setup of your project:
  - 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`.

1.2 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*: Provides access to data files for use as inputs to a workflow, and for comparing to workflow output.
- *workflow_runner*: 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* 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 file.

1.3 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 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 tyep that is loaded as a Python dict and then maps to a user-defined type (struct) in WDL:

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

1.3.1 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 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*.

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):

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

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:

```
{
  "bam": {
    "url": "http://example.com/my.bam",
    "http_headers": {
      "auth_token": "TOKEN"
    }
  },
  "reference": {
    "path": "${REFERENCE_DIR}/chr22.fa"
  },
  "sample": {
    "path": "samples.vcf",
    "contents": "sample1\nsample2"
  },
  "output_vcf": {
    "name": "output.vcf",
    "type": {
      "name": "vcf",
      "allowed_diff_lines": 2
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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 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* 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.
  - `http_headers`: Optional dict mapping header names to values. These headers are used for file download requests. Keys are header names and values are either strings (environment variable name) or mappings with the following keys:
    - `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.

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*. 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 *Installing Data Type Plugins*

1.4 Executors

An Executor is a wrapper around a WDL workflow execution engine that prepares inputs, runs the tool, captures outputs, and handles errors. Currently, Cromwell and Miniwdl are supported, but aternative executors can be implemented as *plugins*.

The `workflow_runner` fixture is a callable that runs the workflow using the executor. It takes one required arguments and some additional optional arguments:

- `wdl_script`: Required; the WDL script to execute. The path may be absolute or relative - if relative, it is first searched relative to the current `tests` directory (i.e. `test_context_dir/tests`), and then the project root.
- `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 is an optional parameter and 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.

You can also pass executor-specific keyword arguments.

1.4.1 Executor-specific options

Cromwell

- `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, or the path to the imports zip file to write, instead of a temp file. By default, all WDL files under the test context directory are imported if an `import_paths.txt` file is not provided.
- `java_args`: Override the default Java arguments.
- `cromwell_args`: Override the default Cromwell arguments.

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`.
- `inputs_file`: Specify the inputs.json file to use, or the path to the inputs.json file to write, instead of a temp file.

1.5 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.

1.5.1 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:

1.5.2 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`):

```
# 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`. Here is an example.

The available configuration options are listed in the following 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.

```
{
  "proxies": {
    "http": {
      "env": "HTTP_PROXY"
    },
    "https": {
      "value": "https://foo.com/proxy",
      "env": "HTTPS_PROXY"
    }
  }
}
```

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*, 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.

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

Executor-specific configuration

Cromwell

Fixtures

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

1.6 Plugins

pytest-wdl provides the ability to implement 3rd-party 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).

1.6.1 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 should 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` to warn about this and add the extra dependencies under the setup.py's `extras_require`. For example:

```
# plugin.py
try:
    import mylib
except ImportError:
    logger.warning(
        "mytype is not available because the mylib library is not "
        "installed"
    )
```

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

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

1.6.2 Creating new executors

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

Your plugin should subclass `pytest_wdl.executors.Executor` and implement the `run_workflow()` method.

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*). For example:

```
setup(
    ...,
    entry_points={
        "pytest_wdl.executors": [
            "myexec = pytest_wdl.executors.myexec:MyExecutor"
        ]
    },
    extras_require={
        "myexec": ["mylib"]
    }
)
```

1.6.3 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 should 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*). For example:

```
setup(
    ...,
    entry_points={
        "pytest_wdl.url_schemes": [
            "myexec = pytest_wdl.url_schemes.myscheme:MyUrlScheme"
        ]
    },
    extras_require={
        "myexec": ["mylib"]
    }
)
```

CHAPTER

TWO

PYTEST_WDL

2.1 pytest_wdl package

2.1.1 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**

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: bool = True*, *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, hash_fn: Callable[[bytes], _hashlib.HASH] = <built-in function openssl_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 module

class pytest_wdl.executors.cromwell.**CromwellExecutor**(*import_dirs: Optional[List[pathlib.Path]] = None*, *java_bin: Union[str, pathlib.Path, None] = None*, *java_args: Optional[str] = None*, *cromwell_jar_file: Union[str, pathlib.Path, None] = None*, *cromwell_config_file: Union[str, pathlib.Path, None] = None*, *cromwell_args: Optional[str] = None*)

Bases: *pytest_wdl.executors.Executor*

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*.

static get_cromwell_outputs(*output*) → dict

get_workflow_imports(*imports_file: Optional[pathlib.Path] = None*) → pathlib.Path

Creates a ZIP file with all WDL files to be imported.

Parameters **imports_file** – Text file naming import directories/files - one per line.

Returns Path to the ZIP file.

run_workflow(*wdl_path: Union[str, 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

- **`Exception`** – 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[List[pathlib.Path]] = None*)

Bases: *pytest_wdl.executors.Executor*

Manages the running of WDL workflows using 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: 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

pytest_wdl.executors.miniwdl.**log_source**(*logger: logging.Logger*, *exn*)

Module contents

class pytest_wdl.executors.**Executor**(*import_dirs: Optional[List[pathlib.Path]] = None*)

Bases: `object`

Base class for WDL workflow executors.

Parameters **`import_dirs`** – Relative or absolute paths to directories containing WDL scripts that should be available as imports.

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

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

pytest_wdl.executors.**compare_output_values**(*expected_value*, *actual_value*, *name: str*) → None

Compare two values and raise an error if they are not equal.

Parameters

- **expected_value** –
- **actual_value** –
- **name** – Name of the output being compared

Raises **AssertionError** –

pytest_wdl.executors.**get_workflow_inputs**(*inputs_dict: Optional[dict] = None*, *inputs_file: Optional[pathlib.Path] = None*, *namespace: Optional[str] = None*) → Tuple[dict, pathlib.Path]

Persist workflow inputs to a file, or load workflow inputs from a file.

Parameters

- **inputs_dict** – Dict of input names/values.
- **inputs_file** – JSON file with workflow inputs.
- **namespace** – Name of the workflow; used to prefix the input parameters when creating the inputs file from the inputs dict.

Returns A tuple (inputs_dict, inputs_file)

pytest_wdl.executors.**make_serializable**(*value*)

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 make serializable.

Returns The serializable value.

pytest_wdl.executors.**validate_outputs**(*outputs: dict*, *expected: dict*, *target: str*) → None

Validate expected and actual outputs are equal.

Parameters

- **outputs** – Actual outputs
- **expected** – Expected outputs

- **target** – Execution target (i.e. workflow name)

Raises **AssertionError** –

pytest_wdl.url_schemes package

Submodules

pytest_wdl.url_schemes.dx module

class pytest_wdl.url_schemes.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*)

class pytest_wdl.url_schemes.dx.**DxUrlHandler**
Bases: *pytest_wdl.url_schemes.UrlHandler*

property handles

property scheme

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

Module contents

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

download_file(*destination: pathlib.Path*, *show_progress: bool = False*)

abstract get_content_length() → Optional[int]

abstract read(*block_size: int*)

class pytest_wdl.url_schemes.**Method**(*src_attr*, *dest_pattern*)
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*)

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**()

2.1.2 Submodules

2.1.3 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*)

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 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.

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.

2.1.4 pytest_wdl.core module

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

class pytest_wdl.core.**DataDirs**(*basedir: pathlib.Path*, *module*, *function: Callable*, *cls: Optional[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.utils.PluginFactory object>, 'miniwd**
Executor plugin modules from the discovered entry points.

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

2.1.5 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.

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: Union[str, pathlib.Path, None]*) → 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*) → Union[str, pathlib.Path, None]

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.

2.1.6 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.

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*)
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

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

2.1.7 pytest_wdl.utils module

class pytest_wdl.utils.**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.utils.**chdir**(*todir: pathlib.Path*)

Context manager that temporarily changes directories.

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

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 *path* is an existing file) 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.**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.**plugin_factory_map**(*return_type: Type[T], group: Optional[str] = None, entry_points: Optional[Iterable[pkg_resources.EntryPoint]] = None*) → Dict[str, pytest_wdl.utils.PluginFactory[~T][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

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*) → 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.

2.1.8 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

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

H

I

J

L

M

N

O

P

R

S

T

U

V

W