modflow-devtools

Release 1.6.0.dev0

MODFLOW Team

May 15, 2024

INTRODUCTION

The *modflow-devtools* package provides a set of tools for developing and testing MODFLOW 6, FloPy, and related applications.

CHAPTER

ONE

INSTALLATION

1.1 Installing `modflow-devtools` from PyPI

Packages are available on PyPi and can be installed with `pip`:

```
pip install modflow-devtools
```

1.2 Installing `modflow-devtools` from source

To set up a `modflow-devtools` development environment, first clone the repository:

```
git clone https://github.com/MODFLOW-USGS/modflow-devtools.git
```

Then install the local copy as well as testing, linting, and docs dependencies:

```
pip install .
pip install ".[lint, test, docs]"
```

1.3 Using `modflow-devtools` as a `pytest` plugin

Fixtures provided by `modflow-devtools` can be imported into a `pytest` test suite by adding the following to the consuming project's top-level `conftest.py` file:

```
pytest_plugins = ["modflow_devtools.fixtures"]
```

1.4 Installing external model repositories

`modflow-devtools` provides fixtures to load models from external repositories:

- `MODFLOW-USGS/modflow6-examples`
- `MODFLOW-USGS/modflow6-testmodels`
- `MODFLOW-USGS/modflow6-largetestmodels`

By default, these fixtures expect model repositories to live next to (i.e. in the same parent directory as) the consuming project repository. If the repos are somewhere else, you can set the `REPOS_PATH` environment variable to point to their parent directory.

Note: a convenient way to persist environment variables needed for tests is to store them in a `.env` file in the `autotest` folder. Each variable should be defined on a separate line, with format `KEY=VALUE`. The `pytest-dotenv` plugin will then automatically load any variables found in this file into the test process' environment.

1.4.1 Installing test models

The test model repos can simply be cloned — ideally, into the parent directory of the `modflow6` repository, so that repositories live side-by-side:

```
git clone https://github.com/MODFLOW-USGS/modflow6-testmodels.git
git clone https://github.com/MODFLOW-USGS/modflow6-largetestmodels.git
```

1.4.2 Installing example models

First clone the example models repo:

```
git clone https://github.com/MODFLOW-USGS/modflow6-examples.git
```

The example models require some setup after cloning. Some extra Python dependencies are required to build the examples:

```
cd modflow6-examples/etc
pip install -r requirements.pip.txt
```

Then, from the `autotest` folder, run:

```
pytest -v -n auto test_scripts.py --init
```

This will build the examples for subsequent use by the tests. To save time, models will not be run — to run the models too, omit `--init`.

CHAPTER TWO

FIXTURES

Several pytest fixtures are provided to help with testing.

2.1 Keepable temporary directories

Tests often need to exercise code that reads from and/or writes to disk. The test harness may also need to create test data during setup and clean up the filesystem on teardown. Temporary directories are built into pytest via the tmp_path and tmp_path_factory fixtures, however the default temporary directory location varies across platforms and may be inconvenient to access.

modflow-devtools provides a set of fixtures to extend default pytest temporary directory fixtures' behavior with the ability to optionally save test outputs in a location of the user's choice:

- function_tmpdir
- module_tmpdir
- class_tmpdir
- session_tmpdir

When pytest is invoked with a --keep <path> option, files created by tests using any of the above fixtures are saved under the specified path, in subdirectories named according to the test function.

The fixtures are named according to their scope, and are automatically created before test code runs and lazily removed afterwards, subject to the same cleanup procedure used by the default pytest temporary directory fixtures. Functionally they are identical to the pytest-provided fixtures save for the behavior described above.

```python
from pathlib import Path
import inspect

def test_tmpdirs(function_tmpdir, module_tmpdir):
    # function-scoped temporary directory
    assert function_tmpdir.is_dir()
    assert inspect.currentframe().f_code.co_name in function_tmpdir.stem

    # module-scoped temp dir (accessible to other tests in the script)
    assert module_tmpdir.is_dir()

    with open(function_tmpdir / "test.txt", "w") as f1, open(module_tmpdir / "test.txt",
↪"w") as f2:
        f1.write("hello, function")
        f2.write("hello, module")
```

There is also a `--keep-failed <path>` option which preserves outputs only from failing test cases. Note that this variant is only compatible with `function_tmpdir`.

2.2 Loading example models

Fixtures are provided to find models from the MODFLOW 6 example and test model repositories and feed them to test functions. Models can be loaded from:

- `MODFLOW-USGS/modflow6-examples`
- `MODFLOW-USGS/modflow6-testmodels`
- `MODFLOW-USGS/modflow6-largetestmodels`

These models can be requested like any other `pytest` fixture, by adding one of the following parameters to test functions:

- `test_model_mf5to6`: a `Path` to a MODFLOW 2005 model namefile, loaded from the `mf5to6` subdirectory of the `modflow6-testmodels` repository
- `test_model_mf6`: a `Path` to a MODFLOW 6 model namefile, loaded from the `mf6` subdirectory of the `modflow6-testmodels` repository
- `large_test_model`: a `Path` to a large MODFLOW 6 model namefile, loaded from the `modflow6-largetestmodels` repository
- `example_scenario`: a `Tuple[str, List[Path]]` containing the name of a MODFLOW 6 example scenario and a list of paths to its model namefiles, loaded from the `modflow6-examples` repository

See the *installation docs* for more information on installing test model repositories.

2.2.1 Configuration

It is recommended to set the environment variable `REPOS_PATH` to the location of the model repositories on the filesystem. Model repositories must live side-by-side in this location, and repository directories are expected to be named identically to GitHub repositories (the directory may have a `.git` suffix). If `REPOS_PATH` is not configured, `modflow-devtools` assumes tests are being run from an `autotest` subdirectory of the consuming project's root, and model repos live side-by-side with the consuming project. If this guess is incorrect and repositories cannot be found, tests requesting these fixtures will be skipped.

Note: by default, all models found in the respective external repository will be returned by these fixtures. It is up to the consuming project to exclude models if needed. This can be accomplished by:

- custom markers
- *filtering with CLI options*
- manually skipping with `pytest.skip(reason="...")`
- *using model-finding utility functions directly*

2.2.2 Usage

MODFLOW 2005 test models

The `test_model_mf5to6` fixture are each a `Path` to the model's namefile. For example, to load `mf5to6` models from the `MODFLOW-USGS/modflow6-testmodels` repo:

```
def test_mf5to6_model(test_model_mf5to6):
    assert isinstance(test_model_mf5to6, Path)
    assert test_model_mf5to6.is_file()
    assert test_model_mf5to6.suffix == ".nam"
```

This test function will be parametrized with all models found in the `mf5to6` subdirectory of the `MODFLOW-USGS/modflow6-testmodels` repository. Note that MODFLOW-2005 namefiles need not be named `mfsim.nam`.

MODFLOW 6 test models

The `test_model_mf6` fixture loads all MODFLOW 6 models found in the `mf6` subdirectory of the `MODFLOW-USGS/modflow6-testmodels` repository.

```
def test_test_model_mf6(test_model_mf6):
    assert isinstance(test_model_mf6, Path)
    assert test_model_mf6.is_file()
    assert test_model_mf6.name == "mfsim.nam"
```

Because these are MODFLOW 6 models, each namefile will be named `mfsim.nam`. The model name can be inferred from the namefile's parent directory.

Large test models

The `large_test_model` fixture loads all MODFLOW 6 models found in the `MODFLOW-USGS/modflow6-largetestmodels` repository.

```
def test_large_test_model(large_test_model):
    print(large_test_model)
    assert isinstance(large_test_model, Path)
    assert large_test_model.is_file()
    assert large_test_model.name == "mfsim.nam"
```

Example scenarios

The `MODFLOW-USGS/modflow6-examples` repository contains a collection of example scenarios, each with 1 or more models. The `example_scenario` fixture is a `Tuple[str, List[Path]]`. The first item is the name of the scenario. The second item is a list of MODFLOW 6 namefile `Path`s, ordered alphabetically by name, with models generally named as follows:

- groundwater flow models begin with `gwf*`
- transport models begin with `gwt*`

This naming permits models to be run in the order provided, with transport models potentially consuming the outputs of flow models. One possible pattern is to loop over models and run each in a subdirectory of the same top-level working directory.

```python
def test_example_scenario(tmp_path, example_scenario):
    name, namefiles = example_scenario
    for namefile in namefiles:
        model_ws = tmp_path / namefile.parent.name
        model_ws.mkdir()
        # load and run model
        # ...
```

Note: example models must first be built by running `pytest -v -n auto test_scripts.py --init` in `modflow6-examples/autotest` before running tests using the `example_scenario` fixture. See the install docs for more info.

2.2.3 Filtering

External model test cases can be filtered by model name or by the packages the model uses with the `--model` and `--package` command line arguments, respectively.

Filtering by model name

Filtering models by name is functionally equivalent to filtering `pytest` cases with `-k`. (In the former case the filter is applied before test collection, while the latter collects tests as usual and then applies the filter.)

For instance, running the `test_largetestmodels.py` script in the `MODFLOW-USGS/modflow6` repository's `autotest/` folder, and selecting a particular model from the `MODFLOW-USGS/largetestmodels` repository by name:

```
autotest % pytest -v test_largetestmodels.py --collect-only --model test1002_biscqtg_
↪disv_gnc_nr_dev
...
collected 1 item
...
```

Equivalently:

```
autotest % pytest -v test_largetestmodels.py --collect-only -k test1002_biscqtg_disv_gnc_
↪nr_dev
...
collected 18 items / 17 deselected / 1 selected
...
```

The `--model` option can be used multiple times, e.g. `--model <model 1> --model <model 2>`.

Filtering by package

MODFLOW 6 models from external repos can also be filtered by packages used. For instance, to select only large GWT models:

```
autotest % pytest -v --package gwt
```

2.2.4 Utility functions

Model-loading fixtures use a set of utility functions to find and enumerate models. These functions can be imported from `modflow_devtools.misc` for use in other contexts:

- `get_model_paths()`
- `get_namefile_paths()`

These functions are used internally in a `pytest_generate_tests` hook to implement the above model-parametrization fixtures. See `fixtures.py` and/or this project's test suite for usage examples.

CHAPTER THREE

MARKERS

Some broadly useful `pytest` markers are provided.

3.1 Default markers

By default, the following markers are defined for any project consuming `modflow-devtools` as a `pytest` plugin:

- `slow`: tests taking more than a few seconds to complete
- `regression`: tests comparing results from different versions of a program

3.1.1 Smoke testing

Smoke testing is a form of integration testing which aims to exercise a substantial subset of the codebase quickly enough to run often during development. This is useful to rapidly determine whether a refactor has broken any expectations before running slower, more extensive tests.

To run smoke tests, use the `--smoke` (short `-S`) CLI option. For instance:

```
pytest -v -S
```

3.2 Conditionally skipping tests

Several `pytest` markers are provided to conditionally skip tests based on executable availability, Python package environment or operating system.

To skip tests if one or more executables are not available on the path:

```
from shutil import which
from modflow_devtools.markers import requires_exe

@requires_exe("mf6")
def test_mf6():
    assert which("mf6")

@requires_exe("mf6", "mp7")
def test_mf6_and_mp7():
    assert which("mf6")
    assert which("mp7")
```

To skip tests if one or more Python packages are not available:

```
from modflow_devtools.markers import requires_pkg

@requires_pkg("pandas")
def test_needs_pandas():
    import pandas as pd

@requires_pkg("pandas", "shapefile")
def test_needs_pandas():
    import pandas as pd
    from shapefile import Reader
```

To mark tests requiring or incompatible with particular operating systems:

```
import os
import platform
from modflow_devtools.markers import requires_platform, excludes_platform

@requires_platform("Windows")
def test_needs_windows():
    assert platform.system() == "Windows"

@excludes_platform("Darwin", ci_only=True)
def test_breaks_osx_ci():
    if "CI" in os.environ:
        assert platform.system() != "Darwin"
```

Platforms must be specified as returned by `platform.system()`.

Both these markers accept a `ci_only` flag, which indicates whether the policy should only apply when the test is running on GitHub Actions CI.

Markers are also provided to ping network resources and skip if unavailable:

- `@requires_github`: skips if `github.com` is unreachable
- `@requires_spatial_reference`: skips if `spatialreference.org` is unreachable

A marker is also available to skip tests if `pytest` is running in parallel with `pytest-xdist`:

```
from os import environ
from modflow_devtools.markers import no_parallel

@no_parallel
def test_only_serially():
    # https://pytest-xdist.readthedocs.io/en/stable/how-to.html#identifying-the-worker-
↪process-during-a-test.
    assert environ.get("PYTEST_XDIST_WORKER") is None
```

3.3 Aliases

All markers are aliased to imperative mood, e.g. `require_github`. Some have other aliases as well:

- `requires_pkg` -> `require[s]_package`
- `requires_exe` -> `require[s]_program`

CHAPTER FOUR

SNAPSHOT TESTING

Snapshot testing is a form of regression testing in which a "snapshot" of the results of some computation is verified and captured by the developer to be compared against when tests are subsequently run. This is accomplished with `syrupy`, which provides a `snapshot` fixture overriding the equality operator to allow comparison with e.g. `snapshot == result`. A few custom fixtures for snapshots of NumPy arrays are also provided:

- `array_snapshot`: saves an array in a binary file for compact storage, can be inspected programmatically with `np.load()`
- `text_array_snapshot`: flattens an array and stores it in a text file, compromise between readability and disk usage
- `readable_array_snapshot`: stores an array in a text file in its original shape, easy to inspect but largest on disk

By default, tests run in comparison mode. This means a newly written test using any of the snapshot fixtures will fail until a snapshot is created. Snapshots can be created/updated by running pytest with the `--snapshot-update` flag.

4.1 Using snapshot fixtures

To use snapshot fixtures, add the following line to a test file or `conftest.py` file:

```
pytest_plugins = [ "modflow_devtools.snapshots" ]
```

CHAPTER

FIVE

WEB UTILITIES

Some utility functions are provided for common web requests. Most use the GitHub API to query information or download artifacts and assets. See this project's test cases (in particular `test_download.py`) for detailed usage examples.

Note: to avoid GitHub API rate limits when using these functions, it is recommended to set the `GITHUB_TOKEN` environment variable. If this variable is set, the token will be borne on requests sent to the API.

5.1 Queries

The following functions ask the GitHub API for information about a repository. The singular functions generally return a dictionary, while the plural functions return a list of dictionaries, with dictionary contents parsed directly from the API response's JSON. The first parameter of each function is `repo`, a string whose format must be `owner/name`, as appearing in GitHub URLs.

For instance, to retrieve information about the latest executables release, then manually inspect available assets:

```python
from modflow_devtools.download import get_release

release = get_release("MODFLOW-USGS/executables")
assets = release["assets"]
print([asset["name"] for asset in assets])
```

This yields `['code.json', 'linux.zip', 'mac.zip', 'win64.zip']`.

Equivalently, using the `get_release_assets()` function to list the latest release assets directly:

```python
from modflow_devtools.download import get_release_assets

assets = get_release_assets("MODFLOW-USGS/executables")
print([asset["name"] for asset in assets])
```

The `simple` parameter, defaulting to `False`, can be toggled to return a simple dictionary mapping asset names to download URLs:

```python
from pprint import pprint

assets = get_release_assets("MODFLOW-USGS/executables", simple=True)
pprint(assets)
```

This prints:

```
{'code.json': 'https://github.com/MODFLOW-USGS/executables/releases/download/12.0/code.
↪json',
 'linux.zip': 'https://github.com/MODFLOW-USGS/executables/releases/download/12.0/linux.
↪zip',
 'mac.zip': 'https://github.com/MODFLOW-USGS/executables/releases/download/12.0/mac.zip',
 'win64.zip': 'https://github.com/MODFLOW-USGS/executables/releases/download/12.0/win64.
↪zip'}
```

5.2 Downloads

The `download_artifact` function downloads and unzips the GitHub Actions artifact with the given ID to the given path, optionally deleting the zipfile afterwards. The `repo` format is `owner/name`, as in GitHub URLs. For instance:

```python
from modflow_devtools.download import list_artifacts, download_artifact

repo = "MODFLOW-USGS/modflow6"
artifacts = list_artifacts(repo, max_pages=1, verbose=True)
artifact = next(iter(artifacts), None)
if artifact:
    download_artifact(
        repo=repo,
        id=artifact["id"],
        path=function_tmpdir,
        delete_zip=False,
        verbose=False,
    )
```

The `download_and_unzip` function is a more generic alternative for downloading and unzipping files from arbitrary URLs.

For instance, to download a MODFLOW 6.4.1 Linux distribution and delete the zipfile after extracting:

```python
from modflow_devtools.download import download_and_unzip

url = f"https://github.com/MODFLOW-USGS/modflow6/releases/download/6.4.1/mf6.4.1_linux.
↪zip"
download_and_unzip(url, "~/Downloads", delete_zip=True, verbose=True)
```

The function's return value is the `Path` the archive was extracted to.

CHAPTER

SIX

LATEX UTILITIES

The `modflow_devtools.latex` module provides utility functions for building LaTeX tables from arrays.

CHAPTER

SEVEN

OS TAGS

MODFLOW 6, Python3, build servers, and other systems may refer to operating systems by different names. Utilities are provided in the `modflow_devtools.ostags` module to convert between

- the output of `platform.system()`
- GitHub Actions `runner.os` tags
- MODFLOW 6 release asset OS tags

Only Linux, Mac and Windows are supported.

7.1 Tag specification

Python3's `platform.system()` returns "Linux", "Darwin", and "Windows", respectively.

GitHub Actions (e.g. `runner.os` context) use "Linux", "macOS" and "Windows".

MODFLOW 6 release asset names end with "linux", "mac" (Intel), "macarm", "win32", or "win64".

7.2 Getting tags

To get the MODFLOW 6 or GitHub tag for the current OS, use:

- `get_modflow_ostag()`
- `get_github_ostag()`

7.3 Converting tags

Conversion functions are available for each direction:

- `python_to_modflow_ostag(tag)`
- `modflow_to_python_ostag(tag)`
- `modflow_to_github_ostag(tag)`
- `github_to_modflow_ostag(tag)`
- `python_to_github_ostag(tag)`
- `github_to_python_ostag(tag)`

Alternatively:

```
convert_ostag(platform.system(), "py2mf") # prints linux, mac, macarm, win32, or win64
convert_ostag(platform.system(), "py2mf") # prints Linux, macOS, or Windows
```

The second argument specifies the mapping in format `<source>2<target>`, where `<source>` and `<target>` may take values `py`, `mf`, or `gh`.

Note: source and target must be different.

7.4 Getting suffixes

A convenience function is available to get the appropriate binary file extensions for a given operating system, identified by any supported OS tag, or the current operating system if no tag is provided. The return value is a 2-tuple containing the executable and library extensions, respectively.

```
get_binary_suffixes()  # get extensions for current OS
get_binary_suffixes("linux")  # returns ("", ".so")
get_binary_suffixes("mac")  # returns ("", ".dylib")
get_binary_suffixes("win64")  # returns (".exe", ".dll")
```

MFZIPFILE

Python's `ZipFile` doesn't preserve file permissions at extraction time. The `MFZipFile` subclass:

- modifies `ZipFile.extract()` to preserve permissions per the recommendation here
- adds a static `ZipFile.compressall()` method to create a zip file from files and directories
- maintains an otherwise identical API

8.1 compressall

The `compressall` method is a static method that creates a zip file from lists of files and/or directories. It is a convenience method that wraps `ZipFile.write()`, `ZipFile.close()`, etc.

```python
from zipfile import ZipFile
from modflow_devtools.zip import MFZipFile

def test_compressall(function_tmpdir):
    zip_file = function_tmpdir / "output.zip"

    input_dir = function_tmpdir / "input"
    input_dir.mkdir()

    with open(input_dir / "data.txt", "w") as f:
        f.write("hello world")

    MFZipFile.compressall(str(zip_file), dir_pths=str(input_dir))
    assert zip_file.exists()

    output_dir = function_tmpdir / "output"
    output_dir.mkdir()

    ZipFile(zip_file).extractall(path=str(output_dir))
    assert (output_dir / "data.txt").is_file()
```

CHAPTER

NINE

TIMED

There is a `@timed` decorator function available in the `modflow_devtools.misc` module. Applying it to any function prints a (rough) benchmark to `stdout` when the function returns. For instance:

```python
from modflow_devtools.misc import timed

@timed
def sleep1():
    sleep(0.001)

sleep1() # prints e.g. "sleep1 took 1.26 ms"
```

It can also wrap a function directly:

```python
timed(sleep1)()
```

The `timeit` built-in module is used internally, however the timed function is only called once, where by default, `timeit` averages multiple runs.

CHAPTER

TEN

TESTING CI WORKFLOWS LOCALLY

The `act` tool uses Docker to run CI workflows in a simulated GitHub Actions environment. Docker Desktop is required for Mac or Windows and Docker Engine on Linux.

Note: `act` can only run Linux-based container definitions. Mac or Windows workflows or matrix OS entries will be skipped.

With Docker installed and running, run `act -l` from the project root to see available CI workflows. To run all workflows and jobs, just run `act`. To run a particular workflow use `-W`:

```
act -W .github/workflows/commit.yml
```

To run a particular job within a workflow, add the `-j` option:

```
act -W .github/workflows/commit.yml -j build
```

Note: GitHub API rate limits are easy to exceed, especially with job matrices. Authenticated GitHub users have a much higher rate limit: use `-s GITHUB_TOKEN=<your token>` when invoking `act` to provide a personal access token. Note that this will log your token in shell history — leave the value blank for a prompt to enter it more securely.

The `-n` flag can be used to execute a dry run, which doesn't run anything, just evaluates workflow, job and step definitions. See the docs for more.

CHAPTER

ELEVEN

GENERATING TOCS

The doctoc tool generates table of contents sections for markdown files.

11.1 Installing Node.js, npm and `doctoc``

doctoc is distributed with the Node Package Manager. Node is a JavaScript runtime environment.

On Ubuntu, Node can be installed with:

```
sudo apt update
sudo apt install nodejs
```

On Windows, with Chocolatey:

```
choco install nodejs
```

Installers and binaries for Windows and macOS are available for download.

Once Node is installed, install doctoc with:

```
npm install -g doctoc
```

11.2 Using doctoc

Then TOCs can be generated with `doctoc <file>`, e.g.:

```
doctoc DEVELOPER.md
```

This will insert HTML comments surrounding an automatically edited region, in which doctoc will create an appropriately indented TOC tree. Subsequent runs are idempotent, scanning for headers and only updating the TOC if the file header structure has changed.

To run doctoc for all markdown files in a particular directory (recursive), use `doctoc some/path`.

By default doctoc inserts a self-descriptive comment

> **Table of Contents** *generated with DocToc*

This can be removed (and other content within the TOC region edited) — doctoc will not overwrite it, only the table.

CHAPTER TWELVE

INDICES AND TABLES

- genindex
- modindex
- search