deltacode

AboutCode.org authors and contributors

Jul 21, 2023

CONTENTS:

Welcome to DeltaCode Documentation!

CHAPTER

ONE

DELTACODE DOCUMENTATION

Welcome to the DeltaCode Documentation.

1.1 Comprehensive Installation

There are multiple ways to install DeltaCode.

- *Using the docker image for testing DeltaCode*

  An alternative to installing the latest DeltaCode release natively is to build a Docker image from the included Dockerfile. The only prerequisite is a working Docker installation.

- *Installation from Source Code: Git Clone*

  You can clone the git source code repository and then run the configure script to configure and install DeltaCode for local and development usage.

- *Installation as a library: via pip*

  To use DeltaCode as a library in your application, you can install it via `pip`. This is recommended for developers or users familiar with Python that want to embed DeltaCode as a library.

1.1.1 Before Installing

- DeltaCode requires a Python version 3.8, 3.9 or 3.10 and is tested on Linux, macOS, and Windows. It should work fine on FreeBSD.

1.1.2 System Requirements

- Hardware : DeltaCode will run best with a modern X86 64 bits processor and at least 8GB of RAM and 2GB of disk space. These are minimum requirements.
- Supported operating systems: DeltaCode should run on these 64-bit OSes running X86_64 processors:
  1. Linux: on recent 64-bit Linux distributions,
  2. Mac: on recent x86 64-bit macOS (10.15 and up, including 11 and 12), Use the X86 emulation mode on Apple ARM M1 CPUs.
  3. Windows: on Windows 10 and up,
  4. FreeBSD.

1.1.3 Prerequisites

DeltaCode needs a Python 3.8 (or above) interpreter.

- **On Linux**:

  Use your package manager to install atleast `python3.8`. If Python 3.8 is not available from your package manager, you must compile it from sources.

  For instance, visit https://github.com/dejacode/about-code-tool/wiki/BuildingPython27OnCentos6 for instructions to compile Python from sources on Centos.

- **On Windows**:

  Download Python from this url: https://www.python.org/

  Install Python on the `c:` drive and use all default installer options. See the Windows installation section for more installation details.

- **On Mac**:

  The default Python 3 provided with macOS is 3.8. Alternatively you can download and install Python 3.8+ from https://www.python.org/

1.1.4 Installation on Linux and Mac

Download and extract the latest DeltaCode release from: https://github.com/nexB/deltacode/releases/latest

Check whether the *Prerequisites* are installed. Open a terminal in the extracted directory and run:

```
./deltacode --help
```

This will configure DeltaCode and display the command line help.

1.1.5 Installation on Windows

Download the latest DeltaCode release zip file from: https://github.com/nexB/deltacode/releases/latest

- In the File Explorer, select the downloaded DeltaCode zip and right-click.
- In the pop-up menu select 'Extract All. . . '
- In the pop-up window 'Extract Compressed (Zipped) Folders' use the default options to extract.
- Once the extraction is complete, a new File Explorer window will pop up.
- In this Explorer window, select the new folder that was created and right-click.

Note: On Windows 10, double-click the new folder, select one of the files inside the folder (e.g., 'setup.py'), and right-click.

- In the pop-up menu select 'Properties'.
- In the pop-up window 'Properties', select the Location value. Copy this to the clipboard and close the 'Properties' window.
- Press the start menu button, click the search box or search icon in the taskbar.
- In the search box type:

```
cmd
```

- Select 'cmd.exe' or 'Command Prompt' listed in the search results.
- A new 'Command Prompt'pops up.
- In this window (aka a 'command prompt'), type 'cd' followed by a space and then Right-click in this window and select Paste. This will paste the path you copied before and is where you extracted DeltaCode:

```
cd path/to/extracted/deltacode
```

- Press Enter.
- This will change the current location of your command prompt to the root directory where DeltaCode is installed.
- Then type:

```
deltacode -h
```

- Press enter. This first command will configure your DeltaCode installation. Several messages are displayed followed by the DeltaCode command help.
- The installation is complete.

1.1.6 Un-installation

- Delete the directory in which you extracted DeltaCode.
- Delete any temporary files created in your system temp directory under a DeltaCode directory.

1.1.7 Using the docker image for testing DeltaCode

- In the project root directory run *docker-compose up*.
- This will create an image of DeltaCode with the name *delta_code*.
- To verify the image created run *docker image ls*.
- To run the image run *docker run -itd –name <specific name of container> delta_code*.
- The above command runs the image in the background and creates a container with the name as per specified.
- To execute the container in a bash mode run *docker exec -it <container name> bash*.
- The above command will open a bash shell in the container.
- To run the commands / pytest inside the shell you can use the commands as specified in the documentations.

1.1.8 Installation from Source Code: Git Clone

You can download the DeltaCode Source Code and build from it yourself. This is what you would want to do it if:

- You are developing DeltaCode or adding new patches or want to run tests.
- You want to test or run a specific version/checkpoint/branch from the version control.

Download the DeltaCode Source Code

Run the following once you have Git installed:

```
git clone https://github.com/nexB/deltacode.git
cd deltacode
```

Configure the build

DeltaCode use a configure scripts to create an isolated virtual environment, install required packaged dependencies.

On Linux/Mac:

- Open a terminal
- cd to the clone directory
- run `./configure`
- run `source venv/bin/activate`

On Windows:

- open a command prompt
- cd to the clone directory
- run `configure`
- run `venv\Scripts\activate`

Now you are ready to use the freshly configured DeltaCode.

Note: For use in development, run instead `configure --dev`. If your face issues while configuring a previous version, `configure --clean` to clean and reset your environment. You will need to run `configure` again.

1.1.9 Installation as a library: via `pip`

DeltaCode can be installed from the public PyPI repository using `pip` which the standard Python package management tool.

The steps are:

1. Create a Python virtual environment:

   ```
   /usr/bin/python3 -m venv venv
   ```

For more information on Python virtualenv, visit this page.

1. Activate the virtual environment you just created:

   ```
   source venv/bin/activate
   ```

2. Run pip to install the latest versions of base utilities:

```
pip install --upgrade pip setuptools wheel
```

3. Install the latest version of DeltaCode:

```
pip install deltacode
```

To uninstall, run:

```
pip uninstall deltacode
```

1.2 Deltacode Output: Format, Fields and Structure

```
Usage: deltacode [OPTIONS]

  Identify the changes that need to be made to the 'old' scan file (-o or --old)
  in order to generate the 'new' scan file (-n or --new).  Write the results to
  a .json file (-j or --json-file) at a user-designated location.  If no file
  option is selected, print the JSON results to the console.

Options:
  -h, --help                Show this message and exit.
  --version                 Show the version and exit.
  -n, --new PATH            Identify the path to the "new" scan file [required]
  -o, --old PATH            Identify the path to the "old" scan file [required]
  -j, --json-file FILENAME  Identify the path to the .json output file
  -a, --all-delta-types     Include unmodified files as well as all changed
                            files in the .json output.  If not selected, only
                            changed files are included.
```

1.2.1 Output Formats

DeltaCode provides two output formats for the results of a DeltaCode codebase comparison: `JSON` and `CSV`.

The default output format is `JSON`. If the command-line input does not include an output flag (`-j` or `--json-file`) and the path to the output file, the results of the DeltaCode comparison will be displayed in the console in `JSON` format. Alternatively, the results will be saved to a `.json` file if the user includes the `-j` or `--json-file` flag and the output file's path, e.g.:

```
deltacode -n [path to the 'new' codebase] -o [path to the 'old' codebase] -j [path to␣
↪the JSON output file]
```

Once a user has generated a DeltaCode JSON output file, he or she can convert that `JSON` output to `CSV` format by running a command with this structure::

```
python etc/scripts/json2csv.py [path to the JSON input file] [path to the CSV output␣
↪file]
```

See also *JSON to CSV Conversion*.

1.2.2 Overall Structure

JSON

Top-Level JSON

DeltaCode's `JSON` output comprises the following six fields/keys and values at the top level:

1. `deltacode_notice` – A string of the terms under which the DeltaCode output is provided.
2. `deltacode_options` – A JSON object containing three key/value pairs:
   - `--new` – A string identifying the path to the `JSON` file containing the ScanCode output of the codebase the user wants DeltaCode to treat as the 'new' codebase.
   - `--old` – A string identifying the path to the JSON file containing the ScanCode output of the codebase the user wants DeltaCode to treat as the 'old' codebase.
   - **`--all-delta-types` – A `true` or `false` value.**
     - This value will be true if the command-line input includes the `-a` or `--all-delta-types` flag, in which case the deltas field described below will include details for unmodified files as well as all changed files.
     - If the user does not include the `-a` or `--all-delta-types` flag, the value will be false and unmodified files will be omitted from the DeltaCode output.
3. `deltacode_version` – A string representing the version of DeltaCode on which the codebase comparison was run.
4. `deltacode_errors` – A list of one or more strings identifying errors (if any) that occurred during the codebase-comparison process.
5. `deltas_count` – An integer representing the number of 'Delta' objects – the file-level comparisons of the two codebases (discussed in the next section) – contained in the DeltaCode output's `deltas` key/value pair.
   - If the user's command-line input does not include the `-a` or `--all-delta-types` flag (see the discussion above of the `--all-delta-types` field/key), the DeltaCode output will omit details for unmodified files and consequently the deltas_count field will not include unmodified files.
6. `deltas` – A list of 'Delta' objects, each of which represents a file-level comparison (i.e., the "delta") of the 'new' and 'old' codebases. The Delta object is discussed in further detail in the next section.

This is the top-level `JSON` structure of the key/value pairs described above:

```
{
  "deltacode_notice": "",
  "deltacode_options": {
    "--new": "",
    "--old": "",
    "--all-delta-types": false
  },
  "deltacode_version": "",
  "deltacode_errors": [],
  "deltas_count": 0,
  "deltas": [one or more Delta objects]
}
```

The Delta Object

Each Delta object consists of four key/value pairs:

- `factors`: A list of one or more strings representing the factors that characterize the file-level comparison and are used to calculate the resulting score, e.g.

  ```
  "factors": [
        "added",
        "license info added",
        "copyright info added"
      ],
  ```

The possible values for the factors field are discussed in some detail in DeltaCode Scoring *Deltacode Scoring*.

- `score`: An integer representing the magnitude/importance of the file-level change – the higher the `score`, the greater the change. For further details about the DeltaCode scoring system, see DeltaCode Scoring *Deltacode Scoring*.
- `new`: A 'File' object containing key/value pairs of certain ScanCode-based file attributes (`path`, `licenses`, `copyrights` etc.) for the file in the codebase designated by the user as `new`. If the Delta object represents the removal of a file (the `factors` value would be `removed`), the value of `new` will be `null`.
- `old`: A 'File' object containing key/value pairs of certain ScanCode-based file attributes for the file in the codebase designated by the user as `old`. If the Delta object represents the addition of a file (the `factors` value would be `added`), the value of `old` will be `null`.

The JSON structure of a Delta object looks like this::

```
{
  "factors": [],
  "score": 0,
  "new": {
    "path": "",
    "type": "",
    "name": "",
    "size": 0,
    "sha1": "",
    "original_path": "",
    "licenses": [],
    "copyrights": []
  },
  "old": {
    "path": "",
    "type": "",
    "name": "",
    "size": 0,
    "sha1": "",
    "original_path": "",
    "licenses": [],
    "copyrights": []
  }
}
```

The File Object

As you saw in the preceding section, the File object has the following JSON structure::

```
{
  "path": "",
```

(continues on next page)

(continued from previous page)

```
  "type": "",
  "name": "",
  "size": 0,
  "sha1": "",
  "original_path": "",
  "licenses": [],
  "copyrights": []
}
```

A File object consists of eight key/value pairs:

- path: – A string identifying the path to the file in question. In processing the 'new' and 'old' codebases to be compared, DeltaCode may modify the codebases' respective file paths in order to properly align them for comparison purposes. As a result, a File object's path value may differ to some extent from its original_path value (see below).
- type: – A string indicating whether the object is a file or a directory.
- name: – A string reflecting the name of the file.
- size: – An integer reflecting the size of the file in KB.
- sha1: – A string reflecting the file's sha1 value.
- original_path: – A string identifying the file's path as it exists in the codebase, prior to any processing by DeltaCode to modify the path for purposes of comparing the two codebases.
- licenses: – A list of License objects reflecting all licenses identified by ScanCode as associated with the file. This list can be empty.
- copyrights: – A list of Copyright objects reflecting all copyrights identified by ScanCode as associated with the file. This list can be empty.

Example of Detailed JSON output

Here is an example of the detailed DeltaCode output in JSON format displaying one Delta object in the deltas key/value pair – in this case, an excerpt from the JSON output of a DeltaCode comparison of zlib-1.2.11 and zlib-1.2.9::

```
{
  "deltacode_notice": "Generated with DeltaCode and provided on an \"AS IS\" BASIS,␣
↪WITHOUT WARRANTIES\nOR CONDITIONS OF ANY KIND, either express or implied. No content␣
↪created from\nDeltaCode should be considered or used as legal advice. Consult an␣
↪Attorney\nfor any legal advice.\nDeltaCode is a free software codebase-comparison tool␣
↪from nexB Inc. and others.\nVisit https://github.com/nexB/deltacode/ for support and␣
↪download.",
  "deltacode_options": {
    "--new": "C:/scans/zlib-1.2.11.json",
    "--old": "C:/scans/zlib-1.2.9.json",
    "--all-delta-types": false
  },
  "deltacode_version": "1.0.0.post49.e3ff7be",
  "deltacode_errors": [],
  "deltas_count": 40,
  "deltas": [
    {
      "factors": [
        "modified"
```

(continues on next page)

(continued from previous page)

```
      ],
      "score": 20,
      "new": {
        "path": "trees.c",
        "type": "file",
        "name": "trees.c",
        "size": 43761,
        "sha1": "ab030a33e399e7284b9ddf9bba64d0dd2730b417",
        "original_path": "zlib-1.2.11/trees.c",
        "licenses": [
          {
            "key": "zlib",
            "score": 60.0,
            "short_name": "ZLIB License",
            "category": "Permissive",
            "owner": "zlib"
          }
        ],
        "copyrights": [
          {
            "statements": [
              "Copyright (c) 1995-2017 Jean-loup Gailly"
            ],
            "holders": [
              "Jean-loup Gailly"
            ]
          }
        ]
      },
      "old": {
        "path": "trees.c",
        "type": "file",
        "name": "trees.c",
        "size": 43774,
        "sha1": "1a554d4edfaecfd377c71b345adb647d15ff7221",
        "original_path": "zlib-1.2.9/trees.c",
        "licenses": [
          {
            "key": "zlib",
            "score": 60.0,
            "short_name": "ZLIB License",
            "category": "Permissive",
            "owner": "zlib"
          }
        ],
        "copyrights": [
          {
            "statements": [
              "Copyright (c) 1995-2016 Jean-loup Gailly"
            ],
            "holders": [
              "Jean-loup Gailly"
```

(continues on next page)

(continued from previous page)

```
            ]
          }
        ]
      }
    },
    [additional Delta objects if any]
  ]
}
```

CSV

Compared with DeltaCode's JSON output, the CSV output is relatively simple, comprising the following seven fields as column headers, with each row representing one Delta object:

- `Score` – An integer representing the magnitude/importance of the file-level change.
- `Factors` – One or more strings – with no comma or other separators – representing the factors that characterize the file-level comparison and are used to calculate the resulting score.
- `Path` – A string identifying the file's path in the 'new' codebase unless the Delta object reflects a `removed` file, in which case the string identifies the file's path in the 'old' codebase. As noted above, this path may vary to some extent from the file's actual path in its codebase as a result of DeltaCode processing for codebase comparison purposes.
- `Name` – A string reflecting the file's name in the 'new' codebase unless the Delta object reflects a `removed` file, in which case the string reflects the file's name in the 'old' codebase.
- `Type` – A string reflecting the file's type ('file' or 'directory') in the 'new' codebase unless the Delta object reflects a `removed` file, in which case the string reflects the file's type in the 'old' codebase.
- `Size` – An integer reflecting the file's size in KB in the 'new' codebase unless the Delta object reflects a `removed` file, in which case the string reflects the file's size in the 'old' codebase.
- `Old Path` – A string reflecting the file's path in the 'old' codebase if the Delta object reflects a `moved` file. If the Delta object does not involve a `moved` file, this field is empty. As with the `Path` field/column header above, this path may differ to some extent from the file's actual path in its codebase due to DeltaCode processing for codebase comparison purposes.

1.3 Deltacode Scoring

1.3.1 Delta Objects

A File-Level Comparison of Two Codebases

A Delta object represents the file-level comparison (i.e., the "delta") of two codebases, typically two versions of the same codebase, using ScanCode-generated `JSON` output files as input for the comparison process.

Based on how the user constructs the command-line input, DeltaCode's naming convention treats one codebase as the "new" codebase and the other as the "old" codebase::

```
deltacode -n [path to the 'new' codebase] -o [path to the 'old' codebase] [...]
```

Basic Scoring

A DeltaCode codebase comparison produces a collection of file-level Delta objects. Depending on the nature of the file-level change between the two codebases, each Delta object is characterized as belonging to one of the categories listed below. Each category has an associated score intended to convey its potential importance – from a license/copyright compliance perspective – to a user's analysis of the changes between the `new` and `old` codebases.

In descending order of importance, the categories are:

1. `added`: A file has been added to the `new` codebase.
2. `modified`: The file is contained in both the `new` and `old` codebase and has been modified (as reflected, among other things, by a change in the file's `sha1` attribute).
3. `moved`: The file is contained in both the `new` and `old` codebase and has been moved but not modified.
4. `removed`: A file has been removed from the `old` codebase.
5. `unmodified`: The file is contained in both the `new` and `old` codebase and has not been modified or moved.

Note: Files are determined to be Moved by looping thru the *added* and *removed* Delta objects and checking their sha1 values.

The score of a Delta object characterized as `added` or `modified` may be increased based on the detection of license- and/or copyright-related changes. See *License Additions and Changes* and *Copyright Holder Additions and Changes* below.

Delta Object Fields and Values

Each Delta object includes the following fields and values:

- `factors`: One or more strings representing the factors that characterize the file-level comparison and resulting score, e.g., in JSON format::

```
"factors": [
  "added",
  "license info added",
  "copyright info added"
],
```

- `score`: A number representing the magnitude/importance of the file-level change – the higher the score, the greater the change.
- `new`: The ScanCode-based file attributes (`path`, `licenses`, `copyrights` etc.) for the file in the codebase designated by the user as `new`.
- `old`: The ScanCode-based file attributes for the file in the codebase designated by the user as `old`.

Note that an `added` Delta object will have a `new` file but no `old` file, while a `removed` Delta object will have an `old` file but not a `new` file. In each case, the `new` and `old` keys will be present but the value for the missing file will be `null`.

1.3.2 License Additions and Changes

Certain file-level changes involving the license-related information in a Delta object will increase the object's score.

- An `added` Delta object's score will be increased:
  - If the `new` file contains one or more licenses (`factors` will include `license info added`).
  - If the the `new` file contains any of the following Commercial/Copyleft license categories (`factors` will include, e.g., `copyleft` added):
    - 'Commercial'
    - 'Copyleft'
    - 'Copyleft Limited'
    - 'Free Restricted'
    - 'Patent License'
    - 'Proprietary Free'
- A `modified` Delta object's score will be increased:
  - If the `old` file has at least one license and the `new` file has no licenses (`factors` will include `license info removed`).
  - If the `old` file has no licenses and the `new` file has at least one license (`factors` will include `license info added`).
  - If both the `old` file and `new` file have at least one license and the license keys are not identical (e.g., the `old` file includes an `mit` license and an `apache-2.0` license and the `new` file includes only an `mit` license) (`factors` will include `license` change).
  - If any of the Commercial/Copyleft license categories listed above are found in the `new` file but not in the `old` file (`factors` will include, e.g., `proprietary free added`).

1.3.3 Copyright Holder Additions and Changes

- An `added` Delta object's score will be increased if the `new` file contains one or more copyright `holders` (`factors` will include `copyright info added`).
- A `modified` Delta object's score will be increased:
  - If the `old` file has at least one copyright `holder` and the `new` file has no copyright holders (`factors` will include `copyright info removed`).
  - If the `old` file has no copyright `holders` and the `new` file has at least one (`actors` will include `copyright info added`).
  - If both the `old` file and `new` file have at least one copyright `holder` and the `holders` are not identical (`factors` will include `copyright` change).

1.3.4 Moved, Removed and Unmodified

As noted above in Basic Scoring *Basic Scoring*, from a license/copyright compliance perspective, the three least significant Delta categories are `moved`, `removed` and `unmodified`.

In the current version of DeltaCode, each of these three categories is assigned a score of 0, with no options to increase that score depending on the content of the Delta object.

However, it is possible that both `moved` and `removed` will be assigned some non-zero score in a future version. In particular, `removed` could be significant from a compliance viewpoint where, for example, the removal of a file results in the removal of a Commercial/Copyleft license obligation.

1.4 Development

TL;DR:

- Contributions comes as bugs/questions/issues and as pull requests.
- Source code and runtime data are in the /src/ directory.
- Test code and test data are in the /tests/ directory.
- Datasets (inluding licenses) and test data are in /data/ sub-directories.
- We use DCO signoff in commit messages, like Linux does.

See CONTRIBUTING.rst for details: https://github.com/nexB/deltacode/blob/develop/CONTRIBUTING.rst

1.4.1 Code layout and conventions

Source code is in the `src/` directory, tests are in the `tests/` directory. Miscellaneous scripts and configuration files are in the `etc/` directory.

There is one Python package for each major feature under `src/` and a corresponding directory with the same name under `tests` (but this is not a package by design as it would not make sense to have a top level "tests" package which is a name that's too common).

Each test script is named `test_XXXX`; we prefer organizing tests in subclasses of the standard library `unittest` module. But we also use plain functions that are discovered nicely by `pytest`.

When source or tests need data files, we store these in a `data` subdirectory.

We use PEP8 conventions with a relaxed line length that can be up to 90'ish characters long when needed to keep the code clear and readable.

We write tests, a lot of tests, thousands of tests. When finding bugs or adding new features, we add tests. See existing test code for examples which form also a good specification for the supported features.

The tests should pass on Linux 64 bits, Windows 64 bits and on macOS 10.14 and up. We maintain multiple CI loops with Azure (all OSes) at https://dev.azure.com/nexB/deltacode/_build and Appveyor (Windows) at https://ci.appveyor.com/project/nexB/deltacode.

Several tests are data-driven and use data files as test input and sometimes data files as test expectation (in this case using either JSON or YAML files); a large number of copyright, license and package manifest parsing tests are such data-driven tests.

1.4.2 Running tests

DeltaCode comes with over 29,000 unit tests to ensure detection accuracy and stability across Linux, Windows and macOS OSes: we kinda love tests, do we?

We use pytest to run the tests: call the `pytest` script to run the whole test suite. This is installed with the `pytest` package which is installed when you run `./configure --dev`).

If you are running from a fresh git clone and you run `./configure` and then `source venv/bin/activate` the `pytest` command will be available in your path.

Alternatively, if you have already configured but are not in an activated "virtualenv" the `pytest` command is available under `<root of your checkout>/venv/bin/pytest`

(Note: paths here are for POSIX, but mostly the same applies to Windows)

If you have a multiprocessor machine you might want to run the tests in parallel (and faster). For instance: `pytest -n4` runs the tests on 4 CPUs. We typically run the tests in verbose mode with `pytest -vvs -n4`.

See also https://docs.pytest.org for details or use the `pytest -h` command to show the many other options available.

One useful option is to run a select subset of the test functions matching a pattern with the `-k` option, for instance: `pytest -vvs -k tcpdump` would only run test functions that contain the string "tcpdump" in their name or their class name or module name.

Another useful option after a test run with some failures is to re-run only the failed tests with the `--lf` option, for instance: `pytest -vvs --lf` would only run only test functions that failed in the previous run.

1.4.3 Thirdparty libraries and dependencies management

DeltaCode uses the `configure` and `configure.bat` scripts to install a virtualenv , install required packaged dependencies using setuptools and such that DeltaCode can be installed in a repeatable and consistent manner on all OSes and Python versions.

For this we maintain a `setup.cfg` with our direct dependencies with loose minimum version constraints; and we keep pinned exact versions of these dependencies in the `requirements.txt` and `requirements-dev.txt` (for testing and development).

And to ensure that we also all use well known version of the core virtualenv, pip, setuptools and wheel libraries, we use the `virtualenv.pyz` Python zipp app from https://github.com/pypa/get-virtualenv/tree/master/public and store it in the Git repo in the `etc/thirdparty` directory.

DeltaCode app archives should not require network access for installation or configuration of its third-party libraries and dependencies. To enable this, we store bundled thirdparty components and libraries in the `thirdparty` directory of released app archives; this is done at build time. These dependencies are stored as pre-built wheels. These wheels are sometimes built by us when there is no wheel available upstream on PyPI. We store all these prebuilt wheels with corresponding .ABOUT and .LICENSE files in https://github.com/nexB/thirdparty-packages/tree/main/pypi which is published for download at https://thirdparty.aboutcode.org/pypi/

Because this is used by the configure script, all the thirdparty dependencies used in DeltaCode MUST be available there first. Therefore adding a new dependency means requesting a merge/PR in https://github.com/nexB/thirdparty-packages/ first that contains all the recursive dependencies.

There are utility scripts in `etc/release` that can help with the dependencies management process in particular to build or update wheels with native code for multiple OSes (Linux, macOS and Windows) and multiple Python versions (3.7+), which is not a completely simple operation (and requires eventually 12 wheels and one source distribution to be published as we support 3 OSes and 4 Python versions).

Using DeltaCode as a Python library

DeltaCode can be used alright as a Python library and is available as as a Python wheel in Pypi and installed with `pip install deltacode`

1.5 JSON to CSV Conversion

The default output format for a DeltaCode codebase comparison is JSON. If the `-j` or `--json-file` option is included in the `deltacode` command, the output will be written to a `.json` file at the user-designated location. For example:

```
deltacode -n [path to the 'new' codebase] -o [path to the 'old' codebase] -j [path to␣
↪the JSON output file]
```

We have also created an easy-to-use script for users who want to convert their JSON output to CSV format. Located at `etc/scripts/json2csv.py`, the conversion can be run with this command template:

```
python etc/scripts/json2csv.py [path to the JSON input file] [path to the CSV output␣
↪file]
```

1.6 Release Process

1.6.1 Steps to cut a new release:

run bumpversion with major, minor or patch to bump the version in:

```
src/deltacode/__init__.py
setup.py
deltacode.ABOUT
```

Update the CHANGELOG.rst commit changes and push changes to develop:

```
git commit -m "commit message"
git push --set-upstream origin develop
```

merge develop branch in master and tag the release.

```
git checkout master
git merge develop
git tag -a v1.6.1 -m "Release v1.6.1"
git push --set-upstream origin master
git push --set-upstream origin v1.6.1
```

Draft a new release in GitHub, using the previous release blurb as a base. Highlight new and noteworthy changes from the CHANGELOG.rst.

Run etc/release/release.sh locally.

Upload the release archives created in the dist/ directory to the GitHub release page.

Save the release as a draft. Use the previous release notes to create notes in the same style. Ensure that the link to thirdparty source code is present.

Test the downloads.

Publish the release on GitHub

Then build and publish the released wheel on Pypi. For this you need your own Pypi credentials (and get authorized to publish Pypi release: ask @pombredanne) and you need to have the twine package installed and configured.

Build a .whl with `python setup.py bdist_wheel` Run twine with `twine upload dist/<path to the built wheel>` Once uploaded check the published release at https://pypi.python.org/pypi/deltacode/ Then create a new fresh local virtualenv and test the wheel installation with: `pip install deltacode`

1.7 Google Summer of Code 2021 - Final report

1.7.1 Project: Virtual Codebase support in DeltaCode

Pratik Dey <pratikrocks.dey11@gmail.com>

1.7.2 Project Overview

The goal of this proposal is to refactor DeltaCode to use Scancode-Toolkit's Virtual Codebase class. This refactoring will allow DeltaCode to be a library as opposed to only be used as a CLI tool, moreover, this refactor will allow DeltaCode to determine deltas much more effectively in the form of BFS tree scan of the two tree structures unlike indexing the entire codebase.

1.7.3 Main Objectives of the project

- Migrate to using VirtualCodebase from the latest scancode.
- Create DeltaCode documentation on Read The Docs.
- Provide the support for fingerprint plugin for Virtual Codebase.
- Provide the Support for enabling Virtual Codebase to scan files having full root paths as their location.

1.7.4 The Project

- Virtual Codebase Integration with Deltacode
- Removing redundant File and License Objects
- Provided options in deltacode scans
- Added Docker Script for Dockerizing the Deltacode Application and make it platform-independent.
- Add Read the Docs Support to Deltacode.

I have completed all the tasks that were in the scope of this GSoC project.

1.7.5 Pull Requests

- https://github.com/nexB/deltacode/pull/167 [Merged]
- https://github.com/nexB/deltacode/pull/176 [Open]
- https://github.com/nexB/deltacode/pull/171 [Open]
- https://github.com/nexB/deltacode/pull/178 [Open]
- https://github.com/nexB/deltacode/pull/168 [Open]

1.7.6 Links

- Project Details
- Proposal
- ScanCode Toolkit
- DeltaCode

I've had a wonderful time during these three months and have learned plenty of things. I would really like to thank @pombredanne, @steven-esser, and @JonoYang for their constant support throughout the journey. From good job claps to nit-picky constructive code-reviews, I enjoyed every bit of this GSoC project.

I had a wonderful time during the GSOC, I learned a lot of things during this time.I really enjoyed this project. I would really like to thank my mentors @pombredanne, @steven-esser, and @TG1999, and all other About code members who constantly supported me throughout this project.