browser-history

Release 0.3.3

Samyak Sarnayak

Aug 02, 2023

CONTENTS:

`browser-history` is a simple, zero-dependencies, developer-friendly python package to retrieve (almost) any browser's history on (almost) any platform.

Check out the *Quick Start* to get started.

This project relies on contributions made by other people especially for browsers other than Chrome, Firefox and Safari and for platforms other than Windows, Mac and Linux. As such, if you notice any issues or if there is no support for your browser or platform, please open an issue on the GitHub Page

CHAPTER

ONE

QUICK START

1.1 Installation

To install the latest release:

```
pip install browser-history
```

To install from the master branch (warning: development version. Things could break):

```
pip install git+https://github.com/browser-history/browser-history.git
```

1.2 Get started

1.2.1 History

To get history from all installed browsers:

```
from browser_history import get_history

outputs = get_history()

# his is a list of (datetime.datetime, url) tuples
his = outputs.histories
```

If you want history from a specific browser:

```
from browser_history.browsers import Firefox

f = Firefox()
outputs = f.fetch_history()

# his is a list of (datetime.datetime, url) tuples
his = outputs.histories
```

- `Firefox` in the above snippet can be replaced with any of the *Supported Browsers*.

1.2.2 Bookmarks

> **Warning:** Experimental feature. Although this has been confirmed to work on multiple (Firefox and Chromium based) browsers on all platforms, it has not been tested as much as the history feature.

To get bookmarks from all installed browsers:

```
from browser_history import get_bookmarks

outputs = get_bookmarks()

# bms is a list of (datetime.datetime, url, title, folder) tuples
bms = outputs.bookmarks
```

To get bookmarks from a specific browser:

```
from browser_history.browsers import Firefox

f = Firefox()
outputs = f.fetch_bookmarks()

# bms is a list of (datetime.datetime, url, title, folder) tuples
bms = outputs.bookmarks
```

1.3 Command Line

Running `browser-history` in shell/terminal/command prompt will return history from all browsers with each line in the output containing the timestamp and URL separated by a comma.

To get history from a specific browser:

```
browser-history -b Firefox
```

Checkout the *Command Line Interface* help page for more information

CHAPTER

TWO

USAGE

This section details all the uses of `browser-history` with complete code examples for each. Refer the *Quick Start* for basic usage.

2.1 Using the API

`browser-history` has an API that allows you to extract history and bookmarks through python code, perhaps in other python programs.

2.1.1 History

The main use case of `browser-history` is to extract browsing history from various browsers installed on the system.

History from all browsers

To get consolidated history from all browsers:

```
from browser_history import get_history

outputs = get_history()

# his is a list of (datetime.datetime, url) tuples
his = outputs.histories
```

History from the default browser

> **Warning:** Experimental feature. This only works on Linux and Windows, but not for every browser. (see this PR to check browser and platform support)

Let `browser-history` automatically detect the default browser set in the system:

```
from browser_history.utils import default_browser

BrowserClass = default_browser()

if BrowserClass is None:
```

(continues on next page)

(continued from previous page)

```
    # default browser could not be identified
    print("could not get default browser!")
else:
    b = BrowserClass()
    # his is a list of (datetime.datetime, url) tuples
    his = b.fetch_history().histories
```

History from a specific browser

If you need histories from a specific browser:

```
from browser_history.browsers import Firefox

f = Firefox()
outputs = f.fetch_history()

# his is a list of (datetime.datetime, url) tuples
his = outputs.histories
```

`Firefox` in the above snippet can be replaced with any of the *Supported Browsers*.

History from a specific profile of a browser

`browser-history` can also extract history from one particular profile of a browser. The profile directory is usually quite different across different systems, this workflow is better suited for the command line tool.

Example:

```
from browser_history.browsers import Firefox

b = Firefox()

# this gives a list of all available profile names
profiles_available = b.profiles(b.history_file)

# use the history_profiles function to get histories
# it needs a list of profile names to use
outputs = b.history_profiles([profiles_available[0]])

his = outputs.histories
```

Save histories to a file

Use `outputs.save("filename.ext")` to save histories to a file (`outputs` is obtained from `fetch_history` as shown in the previous examples). `ext` should be one of the supported extensions (`csv`, `json`, `jsonl`, etc.). See *`save()`* for the list of all supported extensions.

Example:

```
from browser_history import get_history

outputs = get_history()
```

(continues on next page)

(continued from previous page)

```
# save as CSV
outputs.save("history.csv")
# save as JSON
outputs.save("history.json")
# override format
outputs.save("history_file", output_format="json")
```

2.1.2 Bookmarks

Warning: Experimental feature. Although this has been confirmed to work on multiple (Firefox and Chromium based) browsers on all platforms, it is has not been tested as much as the history feature.

`browser-history` also supports extracting bookmarks from some browsers.

All of the usage is similar to extracting history (including saving to a file). You can use the same code examples from *History* with the following changes:

1. Replace `fetch_history` with `fetch_bookmarks` (and `get_history` with `get_bookmarks`)
2. Replace `outputs.histories` with `outputs.bookmarks`

Bookmarks (from `outputs.bookmarks`) are a list of `(datetime.datetime, url, title, folder)` tuples.

2.2 Using the CLI

`browser-history` provides a command-line interface that can be accessed by typing `browser-history` in a terminal (in Windows, this will be the CMD command prompt or powershell).

The CLI provides all of the functionality of `browser-history` (please open an issue if any feature is missing from the CLI).

More information about the CLI here: *Command Line Interface*.

CHAPTER

THREE

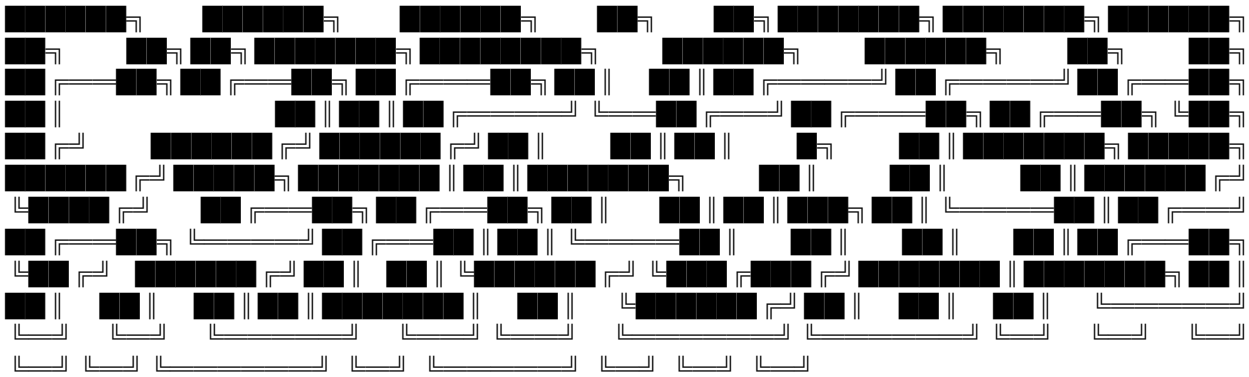
COMMAND LINE INTERFACE

This package provides a command line tool named `browser-history` that is automatically added to path when installed through `pip`.

Help page for the CLI is given below. It can also be accessed using `browser-history --help`.

3.1 CLI Help Page

A tool to retrieve history from (almost) any browser on (almost) any platform



```
usage: browser-history [-h] [-t TYPE] [-b BROWSER] [-f FORMAT] [-o OUTPUT]
                       [-p PROFILE] [--show-profiles BROWSER] [-v]
```

3.1.1 Named Arguments

-t, --type argument to decide whether to retrieve history or bookmarks. Should be one of history, bookmarks. Default is history.

Default: “history”

-b, --browser browser to retrieve history or bookmarks from. Should be one of all, default, Chromium, Chrome, Edge, Opera, OperaGX, Brave, Vivaldi, Firefox, LibreWolf, Safari. Default is all (gets history or bookmarks from all browsers).

Default: “all”

-f, --format	Format to be used in output. Should be one of csv, json, jsonl. Default is infer (format is inferred from the output file's extension. If no output file (-o) is specified, it defaults to csv)
	Default: "infer"
-o, --output	File where history output or bookmark output is to be written. If not provided, standard output is used.
-p, --profile	Specify the profile from which to fetch history or bookmarks. If not provided all profiles are fetched
--show-profiles	List all available profiles for a given browser where browser can be one of default, Chromium, Chrome, Edge, Opera, OperaGX, Brave, Vivaldi, Firefox, LibreWolf, Safari. The browser must always be provided.
-v, --version	show program's version number and exit

Checkout the GitHub repo https://github.com/browser-history/browser-history if you have any issues or want to help contribute

CHAPTER

FOUR

SUPPORTED BROWSERS

This module defines all supported browsers and their functionality.

All browsers must inherit from *browser_history.generic.Browser*.

class browser_history.browsers.**Brave**(*plat=None*)
Brave Browser

Supported platforms:

- Linux
- Mac OS
- Windows

Profile support: Yes

class browser_history.browsers.**Chrome**(*plat=None*)
Google Chrome Browser

Supported platforms:

- Windows
- Linux
- Mac OS

Profile support: Yes

class browser_history.browsers.**Chromium**(*plat=None*)
Chromium Browser

Supported platforms (TODO: Mac OS support)

- Linux
- Windows

Profile support: Yes

class browser_history.browsers.**Edge**(*plat=None*)
Microsoft Edge Browser

Supported platforms

- Windows
- Mac OS

Profile support: Yes

class browser_history.browsers.**Firefox**(*plat=None*)
Mozilla Firefox Browser

Supported platforms:

- Windows
- Linux
- Mac OS

Profile support: Yes

bookmarks_parser(*bookmark_path*)
Returns bookmarks of a single profile for Firefox based browsers The returned datetimes are timezone-aware with the local timezone set by default

Parameters **bookmark_path** (*str*) – the path of the bookmark file

Returns a list of tuples of bookmark information

Return type list(tuple(datetime.datetime, str, str, str))

class browser_history.browsers.**LibreWolf**(*plat=None*)
LibreWolf Browser

Supported platforms:

- Linux

Profile support: Yes

class browser_history.browsers.**Opera**(*plat=None*)
Opera Browser

Supported platforms

- Linux, Windows, Mac OS

Profile support: No

class browser_history.browsers.**OperaGX**(*plat=None*)
Opera GX Browser

Supported platforms

- Windows

Profile support: No

class browser_history.browsers.**Safari**(*plat=None*)
Apple Safari browser

Supported platforms:

- Mac OS

Profile support: No

class browser_history.browsers.**Vivaldi**(*plat=None*)
Vivaldi Browser

Supported platforms (TODO: Add Mac OS support)

- Linux
- Windows

Profile support: Yes

CHAPTER

FIVE

API

5.1 Browser Functionality

Currently, all *Supported Browsers* use the same generic class *browser_history.generic.Browser*.

class browser_history.generic.**Browser**(*plat=None*)

A generic class to support all major browsers with minimal configuration.

Currently, only browsers which save the history in SQLite files are supported.

To create a new browser type, the following class variables must be set.

- *name*
- **paths**: A path string, relative to the home directory, where the browsers data is saved. At least one of the following must be set: *windows_path*, *mac_path*, *linux_path*
- *history_file*
- *history_SQL*

These following class variable can optionally be set:

- *bookmarks_file*
- *bookmarks_parser*
- *profile_support*
- *profile_dir_prefixes*
- *_local_tz*
- *aliases*: A tuple containing other names for the browser in lowercase

Parameters **plat** (Optional[*Platform*]) – the current platform. A value of None means the platform will be inferred from the system.

Examples:

```
>>> class CustomBrowser(Browser):
...     name = 'custom browser'
...     aliases = ('custom-browser', 'customhtm')
...     history_file = 'history_file'
...     history_SQL = """
...         SELECT
...             url
...         FROM
```

(continues on next page)

(continued from previous page)

```
...             history_visits
...     """
...     linux_path = 'browser'
...
... vars(CustomBrowser())
{'profile_dir_prefixes': [], 'history_dir': PosixPath('/home/username/browser')}
```

_local_tz: Optional[datetime.tzinfo] = datetime.timezone(datetime.timedelta(0), 'UTC')
Gets a datetime object of the current time as per the users timezone.

aliases: tuple = ()
Gets possible names (lower-cased) used to refer to the browser type. Useful for making the browser detectable as a default browser which may be named in various forms on different platforms. Do not include *name* in this list

bookmarks_file: Optional[str] = None
Name of the (SQLite, JSON or PLIST) file which stores the bookmarks.

bookmarks_parser(*bookmark_path*)
A function to parse bookmarks and convert to readable format.

bookmarks_path_profile(*profile_dir*)
Returns path of the bookmark file for the given `profile_dir`

The `profile_dir` should be one of the outputs from *profiles()*

Parameters **profile_dir** (`pathlib.Path`) – Profile directory (should be a single name, relative to `history_dir`)

Return type `Optional[Path]`

Returns path to bookmark file of the profile

fetch_bookmarks(*bookmarks_paths=None*, *sort=True*, *desc=False*)
Returns bookmarks of all available profiles stored in SQL or JSON or plist.

The returned datetimes are timezone-aware with the local timezone set by default.

The bookmark files are first copied to a temporary location and then queried, this might lead to some additional overhead and results returned might not be the latest if the browser is in use. This is done because the SQlite files are locked by the browser when in use.

Parameters

- **bookmarks_paths** (list(`pathlib.Path`)) – (optional) a list of bookmark files.
- **sort** (*boolean*) – (optional) flag to specify if the output should be sorted. Default value set to True.
- **desc** – (optional) flag to specify asc/desc (Applicable if sort is True) Default value set to False.

Returns Object of class *browser_history.generic.Outputs* with the attribute bookmarks set to a list of (timestamp, url, title, folder) tuples

Return type *browser_history.generic.Outputs*

fetch_history(*history_paths=None*, *sort=True*, *desc=False*)
Returns history of all available profiles stored in SQL.

The returned datetimes are timezone-aware with the local timezone set by default.

The history files are first copied to a temporary location and then queried, this might lead to some additional overhead and results returned might not be the latest if the browser is in use. This is done because the SQlite files are locked by the browser when in use.

Parameters

- **history_paths** (list(pathlib.Path)) – (optional) a list of history files.
- **sort** (*boolean*) – (optional) flag to specify if the output should be sorted. Default value set to True.
- **desc** – (optional) flag to specify asc/desc (Applicable if sort is True) Default value set to False.

Returns Object of class *browser_history.generic.Outputs* with the data member histories set to list(tuple(datetime.datetime, str)). If the browser is not installed, this object will be empty.

Return type *browser_history.generic.Outputs*

abstract property history_SQL: str
SQL query required to extract history from the history_file. The query must return two columns: visit_time and url. The visit_time must be processed using the datetime function with the modifier localtime.

Return type str

history_dir: pathlib.Path
History directory.

abstract property history_file: str
Name of the (SQLite) file which stores the history.

Return type str

history_path_profile(*profile_dir*)
Returns path of the history file for the given profile_dir

The profile_dir should be one of the outputs from *profiles()*

Parameters **profile_dir** (pathlib.Path) – Profile directory (should be a single name, relative to history_dir)

Return type Optional[Path]

Returns path to history file of the profile

history_profiles(*profile_dirs*)
Returns history of profiles given by *profile_dirs*.

Parameters **profile_dirs** (*list(str)*) – List or iterable of profile directories. Can be obtained from *profiles()*

Returns Object of class *browser_history.generic.Outputs* with the data member histories set to list(tuple(datetime.datetime, str))

Return type *browser_history.generic.Outputs*

classmethod is_supported()
Checks whether the browser is supported on current platform

Returns True if browser is supported on current platform else False

Return type boolean

linux_path: Optional[str] = None
browser path on Linux.

mac_path: Optional[str] = None
browser path on Mac OS.

abstract property name: str
A name for the browser. Not used anywhere except for logging and errors.

Return type str

paths(*profile_file*)
Returns a list of file paths, for all profiles.

Return type list(pathlib.Path)

profile_dir_prefixes: Optional[List[Any]] = None
List of possible prefixes for the profile directories. Keep empty to check all subdirectories in the browser path.

profile_support: bool = False
Boolean indicating whether the browser supports multiple profiles.

profiles(*profile_file*)
Returns a list of profile directories. If the browser is supported on the current platform but is not installed an empty list will be returned

Parameters **profile_file** (*str*) – file to search for in the profile directories. This should be either history_file or bookmarks_file.

Return type list(str)

windows_path: Optional[str] = None
browser path on Windows.

5.2 Outputs Class

class browser_history.generic.**Outputs**(*fetch_type*)
Bases: object

A generic class to encapsulate history and bookmark outputs and to easily convert them to JSON, CSV or other formats.

Parameters **fetch_type** – string argument to select history output or bookmarks output

bookmarks: List[Tuple[datetime.datetime, str, str, str]]
List of tuples of Timestamp, URL, Title, Folder.

field_map: Dict[str, Dict[str, Any]]
Dictionary which maps fetch_type to the respective variables and formatting fields.

format_map: Dict[str, Callable]
Dictionary which maps output formats to their respective functions.

formatted(*output_format='csv'*)
Returns history or bookmarks as a str formatted as output_format

Parameters **output_format** (str) – One the formats in *csv*, *json*, *jsonl*

Return type str

histories: List[Tuple[datetime.datetime, str]]
List of tuples of Timestamp & URL

save(*filename*, *output_format='infer'*)
Saves history or bookmarks to a file. Infers the type from the given filename extension. If the type could not be inferred, it defaults to csv.

Parameters

- **filename** – the name of the file.
- **output_format** – (optional)One the formats in *csv*, *json*, *jsonl*. If not given, it will automatically be inferd from the file's extension

sort_domain()
Returns the history/bookamarks sorted according to the domain-name.

Examples:

```
>>> from datetime import datetime
... from browser_history import generic
... entries = [
...     [datetime(2020, 1, 1), 'https://google.com'],
...     [datetime(2020, 1, 1), 'https://google.com/imghp?hl=EN'],
...     [datetime(2020, 1, 1), 'https://example.com'],
... ]
... obj = generic.Outputs('history')
... obj.histories = entries
... obj.sort_domain()
defaultdict(<class 'list'>, {
    'example.com': [
        [
            datetime.datetime(2020, 1, 1, 0, 0),
            'https://example.com'
        ]
    ],
    'google.com': [
         [
            datetime.datetime(2020, 1, 1, 0, 0),
            'https://google.com'
         ],
         [
            datetime.datetime(2020, 1, 1, 0, 0),
            'https://google.com/imghp?hl=EN'
        ]
    ]
 })
```

Return type DefaultDict[Any, List[Any]]

to_csv()
Return history or bookmarks formatted as a comma separated string with the first row having the fields names

Return type str

Returns string with the output in CSV format

Examples:

```
>>> from datetime import datetime
... from browser_history import generic
... entries = [
...     [datetime(2020, 1, 1), 'https://google.com'],
...     [datetime(2020, 1, 1), 'https://example.com'],
... ]
... obj = generic.Outputs('history')
... obj.histories = entries
... print(obj.to_csv())
Timestamp,URL
2020-01-01 00:00:00,https://google.com
2020-01-01 00:00:00,https://example.com
```

to_json (*json_lines=False*)

Return history or bookmarks formatted as a JSON or JSON Lines format names. If `json_lines` flag is *True* convert to JSON Lines format, otherwise convert it to Plain JSON format.

Parameters **json_lines** (`bool`) – flag to specify if the json_string should be JSON Lines.

Return type `str`

Returns string with the output in JSON/JSONL format

Examples:

```
>>> from datetime import datetime
... from browser_history import generic
... entries = [
...     [datetime(2020, 1, 1), 'https://google.com'],
...     [datetime(2020, 1, 1), 'https://example.com'],
... ]
... obj = generic.Outputs()
... obj.entries = entries
... print(obj.to_json(True))
{"Timestamp": "2020-01-01T00:00:00", "URL": "https://google.com"}
{"Timestamp": "2020-01-01T00:00:00", "URL": "https://example.com"}
>>> print(obj.to_json())
{
    "history": [
        {
            "Timestamp": "2020-01-01T00:00:00",
            "URL": "https://google.com"
        },
        {
            "Timestamp": "2020-01-01T00:00:00",
            "URL": "https://example.com"
        }
    ]
}
```

5.3 Helpers

This module defines some helper code used internally by the `browser_history` package.

Module defines Platform class enumerates the popular Operating Systems.

class browser_history.utils.**Platform**(*value*)
An enum used to indicate the system's platform

A value of 0 is reserved for unknown platforms.

Usage: To be used without instantiating like so:

```
linux = Platform.LINUX
mac = Platform.MAC
windows = Platform.WINDOWS
```

See *get_platform()* to infer the platform from the system.

browser_history.utils.**default_browser**()
This method gets the default browser of the current platform

Returns A *browser_history.generic.Browser* object representing the default browser in the current platform. If platform is not supported or default browser is unknown or unsupported `None` is returned

Return type union[*browser_history.generic.Browser*, None]

browser_history.utils.**get_browser**(*browser_name*)
This method returns the browser class from a browser name.

Parameters **browser_name** – a string representing one of the browsers supported or `default` (to fetch the default browser).

Returns A browser class which is a subclass of *browser_history.generic.Browser* otherwise `None` if no supported browsers match the browser name given or the given browser is not supported on the current platform

Return type union[*browser_history.generic.Browser*, None]

browser_history.utils.**get_browsers**()
This method provides a list of all browsers implemented by browser_history.

Returns A `list` containing implemented browser classes all inheriting from the super class *browser_history.generic.Browser*

Return type `list`

browser_history.utils.**get_platform**()
Returns the current platform

Return type *Platform*

browser_history.utils.**get_platform_name**(*plat=None*)
Returns human readable name of the current platform

Return type `str`

CHAPTER

SIX

CONTRIBUTING

6.1 Development Dependencies

1. Python 3 (versions 3.6 to 3.8 are currently supported)
2. `pip install pylint pytest pytest-cov pre-commit python-dateutil`
   - `pylint` to check for errors and to enforce code style.
   - `pytest` to run the tests.
   - `pytest-cov` to check for code coverage.
   - `pre-commit` to automate code style checks.
   - `python-dateutil` to resolve timezone-related issues in the tests.
3. If you're making changes to the documentation, install the documentation dependencies: `pip install -r docs/requirements.txt`.
   - Refer this for a brief introduction to ReST.

6.2 Development Process - Short Version

1. Select an issue to work on, and inform the maintainers.
2. Fork the repository.
3. `git clone` the forked version of the project.
4. Work on the master branch for smaller patches and a separate branch for new features.
5. Initialize pre-commit hook (only on first commit) by running: `pre-commit install`
6. Make changes, `git add` and then commit. Make sure to link the issue number in the commit message.
7. Run the following commands: `pylint browser_history`, `pytest --cov=./browser_history`
8. (Optional) If you're updating the documentation, make sure you update `docs/quickstart.rst` and `README.md` simultaneously. Run the following: `cd docs`, `make html` and then open `_build/html/index.html` in a browser to confirm that the documentation rendered correctly.
9. If all tests are passing, pull changes from the original remote with a rebase, and push the changes to your remote repository.
10. Use the GitHub website to create a Pull Request and wait for the maintainers to review it.

6.3 Development Process - Long Version

1. Select an issue to work on, and inform the maintainers.
   - Look for issues, find something that you want to work on.
   - Leave a comment on the issue saying that you want to work on it. The maintainers will give you the go-ahead.
2. Fork the repository.
   - The fork button will be available on the top right in the GitHub website.
3. `git clone` the forked version of the project.
   - `git clone https://github.com/<your-github-username>/browser-history.git`
   - Add a remote to the original repository and name it `upstream`.
   - `git remote add upstream https://github.com/browser-history/browser-history.git`
4. Work on the master branch for smaller patches and a separate branch for new features.
   - To create a new feature branch and use it, run: `git checkout -b feature-<feature-name>`.
   - If a feature branch already exists, switch to it before committing: `git checkout feature-<feature-name>`
5. Initialize the pre-commit hook, if you are committing changes for the first time:
   - `pre-commit install` - to setup git hook scripts that will help you check your code each time you commit.
6. Make changes, `git add` and then commit. Make sure to link the issue number in the commit message.

   Caution: When testing changes, create and activate a virtual environment and install the package in editable mode using `pip install -e .` to ensure that the pip version is not used.

   - `git add <names of all modified files>`
   - `git commit`
   - Make your commit descriptive. The above command will open your text editor.
   - (optional) Tag the commit with appropriate tags such as: (see `git log` for examples)
     - `[CODE]` - changes to the main code (other than CLI)
     - `[CLI]` - changes to the CLI code
     - `[FIX]` - bug fixes
     - `[TESTS]` - updates to tests
     - `[CI]` - changes related to integrations such as GitHub actions workflows, codecov, etc.
     - `[DOC]` - changes to the documentation
   - Continuing the above theme, it is preferred to add changes to a single part in one commit. For example, changes to the code, tests and docs for the same feature could go into separate commits.
   - Write the commit message on the first line and a short description about your change. Save and quit the editor to commit your change.
7. Run the following commands:

- `pylint browser_history` - ensure that there are no errors (codes starting with an `E`).
- `pytest --cov=./browser_history` - ensure that all tests pass.

8. (Optional) If you're updating the documentation, run the following:

> **Caution:** Update `docs/quickstart.rst` and `README.md` simultaneously.

- Change to the docs directory: `cd docs`
- Build the documentation: `make html`
- Open `_build/html/index.html` in a browser to confirm that the documentation rendered correctly.

9. If all tests are passing, pull changes from the original remote with a rebase, and push the changes to your remote repository.

> **Caution:** If you are working on a feature branch, use that branch name instead of master.

- `git pull --rebase upstream master`
- In the extremely small chance that you run into a conflict, just open the files having the conflict and remove the markers and edit the file to the one you want to push. After editing, run `git rebase --continue` and repeat till no conflict remains.
- Verify that your program passes all the tests, and your change actually works in general.

> **Caution:** If you are working on a feature branch, use that branch name instead of master.

- Push your changes to your fork: `git push origin master`

10. Use the GitHub website to create a Pull Request and wait for the maintainers to review it.

- Visit your forked repository and click on "Pull Request". The Pull Request must always be made to the `browser-history/master` branch. Add the relevant description, ensure that you link the original issue.
- The maintainers will review your code and see if it is okay to merge. It is quite normal for them to suggest you to make some changes in this review.
- If you are asked to make changes, all you need to do is:

```
# make your change
git add <files that you changed>
git commit
git push origin master      # if you are working on a feature branch, use␣
↪that branch name instead of master
```

- The changes are immediately reflected in the pull request. Once the maintainers are satisfied, they will merge your contribution :)

As long as you follow the above instructions things should go well. You are always welcome to ask any questions about the process, or if you face any difficulties in the `#browser-history-help` channel on the PES Open Source Slack .

6.4 Release Overview

(for the more regular contributors)

- `master` branch for development. Small patches/enhancements go here.
- `release` branch for tagged releases. This is the branch that will be shipped to users.
- Separate `feature-x` branches for adding new "big" features. These branches are merged with master, on completion.
- Once we are satisfied with a certain set of features and stability, we pull the changes from master to release. A new release tag is made.
  - The release workflow will automatically submit the release to PyPI. Ensure that version numbers are changed where necessary (`setup.py`, docs, etc.) - PyPI does not accept new files for the same version number, once a version is published it cannot be changed.
- If bugs were found on the stable release, we create a hotfix branch and fix the bug. The master branch must also pull the changes from hotfix. A new release tag is created (incrementing with a smaller number).
  - We follow semantic versioning .

6.5 Code of Conduct

This project follows the PES Open Source Code of Conduct .

6.6 Technicalities

(might be outdated)

6.6.1 Adding support for a new browser

Browsers are defined in `browser_history/browsers.py`. To add a new browser, create a new class extending the *`browser_history.generic.Browser`*. See *Browser Functionality* for the class variables that must be set and their description. Currently only browsers which use SQLite databases to store history are supported.

6.6.2 Adding support for a new platform

(platform here refers to an OS)

This one is tricky. Platforms are defined in *`browser_history.utils.Platform`*. To add a new platform, the following must be updated.

- Add the platform to *`browser_history.utils.Platform`*
- Update *`browser_history.utils.get_platform()`* to correctly return the platform.
- Update the `__init__` method of *`browser_history.generic.Browser`* and create a new class variable for that platform.
- Update as many *Supported Browsers* as possible with the `platform_path` for the new platform.

6.6.3 Tests

Test Home directory:

Tests are done by including a minimal copy of the browser files with the correct paths. For example, on `Linux` platform and for the `Firefox` browser, `tests/test_homedirs/Linux` contains a minimal snapshot of the home directory with only the files required for extracting history which is the following for `Firefox` on `Linux`:

```
test_homedirs/Linux
└── .mozilla
    └── firefox
        └── profile
            └── places.sqlite
```

It would be a great help for us if you contribute any missing platform-browser combination, even if you don't write any tests accompanying them.

Writing Tests:

Tests are executed using pytest. Monkeypatching is used to change the home directory to one of the test directories and to emulate the home directory of a different platform.

The monkeypatches are defined in `tests/utils.py`. The `change_homedir` fixture must be used for all tests and one of `become_windows`, `become_mac` or `become_linux`. Look at some tests in `tests/test_browsers.py` for examples.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L