coinflip

Release 0.1.5

Matthew Barber

Feb 26, 2021

CONTENTS

CHAPTER

ONE

COINFLIP

Randomness testing for humans

coinflip aims to implement the tests recommended by NIST SP800-22 to check random number generators for randomness. A user-friendly command-line interface provided allows you to `run` the tests on your data, and subsequently `report` on the results by generating informational HTML documents.

`coinflip.randtests` acts as the public API for notebook users and developers to use the randomness tests directly. The tests are implemented as general solutions, meaning they accept basically any sequence with two distinct elements!

1.1 Setup

You can get the latest release of coinflip from PyPI.

```
$ pip install coinflip
```

Alternatively you can get the (unstable) development version straight from GitHub.

```
$ pip install git+https://github.com/Honno/coinflip
```

If that means nothing to you, no fret! Please continue reading the instructions below.

1.1.1 Install Python 3.7+

Cross-platform installation instructions for Python are available at realpython.com/installing-python/.

Note `coinflip` only works on **Python 3.7 or above**. Make sure you have Python 3.7 (or higher) by checking the version of your installation:

```
$ python --version
Python 3.7.X
```

1.1.2 Clone repository

You can clone the source code via Git:

```
$ git clone https://github.com/Honno/coinflip
```

1.1.3 Install coinflip

Enter the directory *coinflip* is downloaded to:

```
$ cd coinflip
```

You can install *coinflip* via the `pip` module:

```
$ pip install -e .
```

pip is the standard package manager for Python, which should of installed automatically when installing Python 3.7+.

1.1.4 Trial run

Try running the randomness tests on an automatically generated binary sequence:

```
$ coinflip example-run
```

If the command `coinflip` is "not found", you may need to add your local binaries folder to your shell's path. For example, in bash you would do the following:

```
$ echo "export PATH=~/.local/bin:$PATH" >> ~/.bash_profile
$ source ~/.bash_profile
```

In the worst case, you can execute commands via `python -m`:

```
$ python -m coinflip example-run
```

1.2 Quick start

Randomness tests can be ran over your RNG output via the `run` command.

```
$ coinflip run DATA OUT
...
```

`DATA` is the path to newline-delimited text file that contains a binary sequence. An example file to use is available on my gist. Alternatively, raw binary files can be read as bitstreams via the `--binary` flag

`OUT` is the path where you want the results to be saved. The results will be saved as a pickle-serialised file, which can be viewed again via the `read` command. Additionally you can generate informational HTML reports from the results via the `report` command, but note that the reports are currently very lacking.

Output should comprise of the sequence parsed from `DATA`, test-specific result summaries, and a final overall summary table.

CHAPTER

TWO

COMMANDS

2.1 coinflip

Randomness tests for RNG output.

Randomness tests can be ran on the your binary data via the run command. Informational web documents explaining the test results can be produced via the report command.

```
coinflip [OPTIONS] COMMAND [ARGS]...
```

Commands

example-run
Run randomness tests on automatically…

read
Print test results.

report
Generate an informational web document from…

run
Run randomness tests on DATA and write…

2.2 coinflip run

Run randomness tests on DATA and write results to OUT.

DATA is a newline-delimited text file which contains output of a random number generator.

Individual results of each test are printed as they come. Once they are all finished, the results are written to OUT.

The results saved in OUT can be printed again via the read command. OUT can also be used to generate an informational web document via the report command.

```
coinflip run [OPTIONS] DATA OUT
```

Options

-b, --binary
Read DATA as a raw binary file.

Arguments

DATA
Required argument

OUT
Optional argument

2.3 coinflip example-run

Run randomness tests on automatically generated data.

```
coinflip example-run [OPTIONS]
```

Options

-e, --example `<example>`
Example binary output to use.

Options Python | Mersenne

-n, --length `<length>`
Length of binary output.

-t, --test `<test>`
Specify single test to run on data.

Options monobit | frequency_within_block | runs | longest_runs | binary_matrix_rank | spectral | non_overlapping_template_matching | overlapping_template_matching | maurers_universal | linear_complexity | serial | approximate_entropy | cusum | random_excursions | random_excursions_variant | exceptions

2.4 coinflip read

Print test results.

```
coinflip read [OPTIONS] RESULTS
```

Arguments

RESULTS
Required argument

2.5 coinflip report

Generate an informational web document from results.

```
coinflip report [OPTIONS] RESULTS OUT
```

Arguments

RESULTS
Required argument

OUT
Optional argument

CHAPTER THREE

REFERENCE

3.1 randtests

Statistical tests for randomness

This sub-package comprises of implementations of statistical tests laid out in a comprehensive paper from NIST[1] in regards to assessing (pseudo-)random number generators.

Notes

A copy of the NIST paper can be found at the root of the `coinflip` repository as SP800-22.pdf.

The test themselves are defined in section 2., "Random Number Generation Tests", p. 23-62. Further detail of the tests are provided in section 3. "Technical Descriptions of Tests", p. 63-87.

These tests were implemented in a complimentary program `sts`, which can be downloaded from the NIST website.

Note that the paper assumes a great amount of familiarity with certain concepts in statistics. It also uses some constants and algorithms without any explanations. Part of the purpose for `coinflip` is to "describe" the NIST tests more wholly than in the paper itself, whilst also reducing the noise of some non-idiomatic programming conventions used in `sts`.

`coinflip.randtests.`**`monobit`**(*sequence*)

Proportion of values is compared to expected 1:1 ratio

The difference between the occurrences of the two values in the sequence is found, and referenced to a hypothetically truly random sequence.

Parameters **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements

Returns **result** (`MonobitTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`frequency_within_block`**(*sequence*, *blocksize: Optional[int] = None*)

Proportion of values per block is compared to expected 1:1 ratio

The sequence is split into blocks, and the difference between the occurrences of the two values in the sequence is found. This is referenced to a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements

[1] National Institute of Standards and Technology <Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, Lawrence E Bassham II>, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", *Special Publication 800-22 Revision 1a*, April 2010.

- **blocksize** (`int`) – Size of the blocks that partition the given sequence

Returns result (`FrequencyWithinBlockTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`runs`**`(`*sequence*`)`

Number of runs is compared to expected result

The number of runs (uninterrupted sequence of the same value) is found, and referenced to a hypothetically truly random sequence.

Parameters sequence (*array-like with two distinct values*) – Sequence containing 2 distinct elements

Returns result (`RunsTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`longest_runs`**`(`*sequence*`)`

Longest runs per block is compared to expected result

The sequence is split into blocks, where the longest number of runs (uninterrupted sequence of the same value) in each block is found. This is referenced to a hypothetically truly random sequence.

Parameters sequence (*array-like with two distinct values*) – Sequence containing 2 distinct elements

Returns result (`LongestRunsTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`binary_matrix_rank`**`(`*sequence*, *matrix_dimen: Optional[Tuple[int, int]] = None*`)`

Independence of neighbouring sequences is compared to expected result

The sequence is split into matrices, where the rank of each matrix is found to determine independence. The counts of different rank bins is referenced to a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **matrix_dimen** (`Tuple[int, int]`) – Number of rows and columns in each matrix

Returns result (`BinaryMatrixRankTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`spectral`**`(`*sequence*`)`

Potency of periodic features in sequence is compared to expected result

The sequence is treated as a signal, which is applied a Fourier transform so as to find constituent periodic features. The strength of these features is referenced to the expected periodic features present in a hypothetically truly random sequence.

Parameters sequence (*array-like with two distinct values*) – Sequence containing 2 distinct elements

Returns result (`SpectralTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

Raises `NonBinaryTruncatedSequenceError` – When odd-lengthed sequence is truncated there is only one distinct value present

`coinflip.randtests.`**`non_overlapping_template_matching`**`(`*sequence*, *template_size: Optional[int] = None*, *blocksize: Optional[int] = None*`)`

Matches to template per block is compared to expected result

The sequence is split into blocks, where the number of non-overlapping pattern matches to the template in each block is found. This is referenced to the expected mean and variance in matches of a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **template_size** (`int`) – Size of all the templates to be generated
- **blocksize** (`int`) – Size of the blocks that partition the given sequence

Returns **results** (`NonOverlappingTemplateMatchingMultiTestResult`) – Dataclass that contains the statistics and p-values of all sub-tests, as well as other relevant information gathered.

`coinflip.randtests.`**`overlapping_template_matching`**(*sequence*, *template_size: Optional[int] = None*, *blocksize: Optional[int] = None*, *matches_ceil: Optional[int] = None*)

Overlapping matches to template per block is compared to expected result

The sequence is split into blocks, where the number of overlapping patterns matches to the template in each block is found. This is referenced to the expected mean and variance in matches of a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **template_size** (`int`) – Size of the template to be generated
- **blocksize** (`int`) – Size of the blocks that partition the given sequence
- **matches_ceil** (`int`) – Group matches of this value and higher as one single tally

Returns **result** (`OverlappingTemplateMatchingTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`maurers_universal`**(*sequence*, *blocksize: Optional[int] = None*, *init_nblocks: Optional[int] = None*)

Distance between patterns is compared to expected result

The distinct patterns in an initial sequence are identified, and the distances between subsequent occurences of these patterns in a remaining sequence are accumulated. The normalised value for the accumulated distances is referenced to a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **blocksize** (`int`) – Size of the patterns
- **init_nblocks** (`int`) – Number of initial blocks to identify patterns

Returns **result** (`UniversalTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`linear_complexity`**(*sequence*, *blocksize: Optional[int] = None*)

LSFRs of blocks is compared to expected length

The sequence is split into blocks, where the shortest linear feedback shift register is found for each block. The difference of each LSFR's length to the expected mean length is binned, and is referenced to a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **blocksize** (`int`) – Size of the blocks

Returns results (`LinearComplexityTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`serial`**(*sequence*, *blocksize: Optional[int] = None*)

Proportion of all overlapping patterns is compared to expected uniformity

The number of overlapping pattern matches for each distinct pattern is found. This is referenced to a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **blocksize** (`int`) – Size of the patterns

Returns results (`SerialMultiTestResult`) – Dataclass that contains the statistics and p-values of all sub-tests, as well as other relevant information gathered.

`coinflip.randtests.`**`approximate_entropy`**(*sequence*, *blocksize: Optional[int] = None*)

Approximate entropy of sequence is compared to expected result

The approximate entropy of the sequence is found and referenced to a truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **blocksize** (`int`) – Size of the patterns

Returns results (`ApproximateEntropyTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`cusum`**(*sequence*, *reverse: bool = False*)

Furthest detour in a randomn walk is compared to expected result

The sequence is treated as a random walk, where the furthest detour from the axis is identified and referenced to a hypothetically truly random sequence.

Parameters

- **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements
- **reverse** (`bool`) – Cumulate sums from the end of the sequence first.

Returns results (`CusumTestResult`) – Dataclass that contains the test's statistic and p-value as well as other relevant information gathered.

`coinflip.randtests.`**`random_excursions`**(*sequence*)

Frequency of states per cycle in a random walk is compared to expected results

The sequence is treated as a random walk, where the frequency of states -4 to 4 in each cycle is found. This is referenced to a hypothetically truly random sequence.

Parameters sequence (*array-like with two distinct values*) – Sequence containing 2 distinct elements

Returns results (`RandomExcursionsMultiTestResult`) – Dataclass that contains the statistics and p-values of all sub-tests, as well as other relevant information gathered.

`coinflip.randtests.`**`random_excursions_variant`**(*sequence*)

Proportion of states in a random walk is compared to expected results

The sequence is treated as a random walk, where the occurrences of states -9 to 9 is founded and referenced to a hypothetically truly random sequence.

Parameters **sequence** (*array-like with two distinct values*) – Sequence containing 2 distinct elements

Returns **results** (`RandomExcursionsVariantMultiTestResult`) – Dataclass that contains the statistics and p-values of all sub-tests, as well as other relevant information gathered.

exception `coinflip.randtests.exceptions.`**`TestError`**
Base class for test-related errors

exception `coinflip.randtests.exceptions.`**`TestNotImplementedError`**
Error if test is not implemented to handle valid parameters

exception `coinflip.randtests.exceptions.`**`TestInputError`**
Error if test cannot handle (invalid) parameters

exception `coinflip.randtests.exceptions.`**`NonBinarySequenceError`**
Error if sequence does not contain only 2 distinct values

3.2 algorithms

Algorithm implementations

`coinflip.algorithms.`**`matrix_rank`**(*matrix: Iterable[Iterable[typing_extensions.Literal[0, 1]]]*) → Union[int, numpy.int64, numpy.int32, numpy.int8, numpy.int16, numpy.uint8, numpy.uint16, numpy.uint32, numpy.uint64]
Finds the rank of a binary matrix

Parameters **matrix** (`Iterable[Iterable[Integer]]`) – Binary matrix to rank

Returns **rank** (*Integer*) – Rank of `matrix`

Notes

Implementaton inpisred by a StackOverflow answer from Mark Dickinson.

`coinflip.algorithms.`**`berlekamp_massey`**(*sequence: Sequence[typing_extensions.Literal[0, 1]]*) → int
Finds the shortest LSFR in a sequence

3.3 collections

Specialised container datatypes

class `coinflip.collections.`**`Bins`**(*intervals: Iterable[numbers.Real]*)
Mapping that initialises intervals as empty bins

If a key is accessed that does not exist, the nearest interval is used.

Parameters **intervals** (`Iterable[Real]`) – Non-existent keys will round to the closest of these intervals

Examples

```
>>> bins = Bins([-6, -3, 0, 3, 6])
>>> bins
{-6: 0, -3: 0, 0: 0, 3: 0, 6: 0}
>>> bins[3] += 1                    # n = 3
>>> bins
{-6: 0, -3: 0, 0: 0, 3: 1, 6: 0}
>>> bins[7] += 1                    # n = 6
>>> bins[11] += 1                   # n = 6
>>> bins[6.5] += 1                  # n = 6
>>> bins
{-6: 0, -3: 0, 0: 0, 3: 1, 6: 3}
>>> bins[-1000000] += 1             # n = -6
>>> bins
{-6: 1, -3: 0, 0: 0, 3: 1, 6: 3}
>>> bins[0.5] += 1                  # n = 0
>>> bins
{-6: 1, -3: 0, 0: 1, 3: 1, 6: 3}
>>> del bins[6]
{-6: 1, -3: 0, 0: 1, 3: 4}
```

class coinflip.collections.**defaultlist**(*default_factory: Optional[Callable] = None*)

A list with default values

Parameters **default_factory** (`Callable`, optional)

class coinflip.collections.**FloorDict**(*dict: Dict*)

Mapping where invalid keys floor to the smallest real key

If a key is accessed that does not exist, the nearest real key that is the less-than of the passed key is used.

Parameters **dict** (`Dict`) – Dictionary containing the key-value pairs to be floored to

3.4 generators

Generators of binary sequences

Objects infinitely generate `0` and `1` integers to represent a binary sequence.

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

coinflip could always use more documentation, whether as part of the official coinflip docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/Honno/coinflip/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

To set up *coinflip* for local development:

1. Fork coinflip (look for the "Fork" button).
2. Clone your fork locally:

   ```
   git clone git@github.com:Honno/coinflip.git
   ```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.
4. When you're done making changes run all the checks and docs builder with tox one command:

   ```
   tox
   ```

5. Commit your changes and push your branch to GitHub:

   ```
   git add .
   git commit -m "Your detailed description of your changes."
   git push origin name-of-your-bugfix-or-feature
   ```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.

It will be slower though …

CHAPTER

FIVE

AUTHORS

- Matthew Barber - https://matthewbarber.io

CHAPTER

SIX

CHANGELOG

6.1 0.1.5 (2020-12-08)

- **`coinflip.collections.defaultlist` improvements:**
  - *index()* and *remove()* uses the *__iter__()* method.
  - Setter for the *default_factory* property.
  - Returns a *defaultlist* on slice get operations.

6.2 0.1.4 (2020-11-28)

- Slicing support for accessor methods in `coinflip.collections.defaultlist`.
- Fixed bug causing inaccurate results for the *Non-overlapping Template Matching* test.
- `Bins` and `FloorDict` containers in `coinflip.collections` are now always ordered.

6.3 0.1.3 (2020-11-19)

- Fixed wrong intervals being rounded to in `coinflip.collections.bins`.

6.4 0.1.2 (2020-11-18)

- `coinflip.collections.bins` now uses `OrderedDict` under the hood.

6.5 0.1.1 (2020-11-16)

- Reference implementation available in `coinflip.randtests_refimpl` (requires the *refimpl* extra).
- Docstrings for all the randomness tests in `coinflip.randtests`.

6.6 0.1.0 (2020-11-09)

- First beta release on PyPI.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX