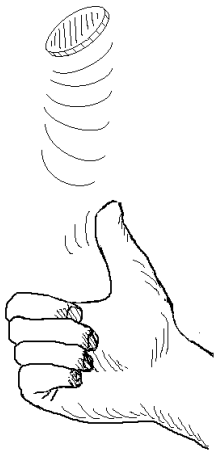
Developing *coinflip*, a Randomness Testing Suite

Matthew Barber

August 2, 2020



Code repository: https://github.com/Honno/coinflip/
Documentation: https://coinflip.readthedocs.io/

Abstract

This report covers the process of creating *coinflip*, a Python library for randomness testing, which also includes a command-line interface.

After the introduction, the following sections detail various aspects of implementing coinflip. A final evaluation is found at the end, with references and appendix items following suit.

Note the code *SP800-22* refers to the NIST paper [1] which was referenced heavily when implementing coinflip. A copy is available at https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final.

Contents

List of Listings

List of Figures

List of Tables

1 Introduction

Random number generators (RNGs) can be used for tasks which require unpredictability, namely cryptographic applications. RNG algorithms can incorporate phenomena that is currently understood to be random as *entropy sources*, such as thermal fluctuations and atom decay. Further assurances should be made on a RNGs' unpredictability however, due to possible erroneous observations of such phenomena or misapplication of observations in the algorithm's logic.

Statistical tests exist to test for randomness for such assurances. These are hypothesis tests, identifying some property of a sample output from the RNG represented in a *test statistic*, which is compared to a hypothetically truly random sequence. A *p-value* results from these tests, suggesting the probability that a truly random sequence would have the characteristics of the RNG output—the lower the p-value, the less confident one can say the RNG is truly random.

Some software suites exist that implement these statistical tests on user-provided generator binary outputs. Two comprehensive and widely-used suites are *TestU01* [2] and National Institute of Standards and Technology's (NIST) *Statistical Test Suite* [1] (referred to as *sts*).

In my own use of these suites' command-line interfaces, I have experienced varying degrees of frustration when running randomness tests over RNG output. In particular, there is exploration to be had on how the process of translating one's RNG output to a machine-readable format could be made easier (3.1).

I also found results of running tests in these suites typically comprising of just the test's statistic and p-value, which makes the usefulness of these suites tied to one's own expertise of how the tests work. Beyond writing more descriptive result messages, the use of graphical charts could be leveraged to educate on the test-specific logic to general statistical theory (2.2.1).

Whilst these suites are open source projects, frustrations in the actual code and architecture of these suites accumulate into significant maintenance issues, preventing such extensions to be easily added. Sometimes the unquestioning use of scientific material itself actively caused confusion (2.1.2), but namely there were suite-specific gripes that led me to write my own.

Python was employed, primarily due to my own familiarity with the language and its ecosystem. Qualities such as its readable syntax and duck typing (2.1.1) allows for code to better express high-level logic. Mature packages bring this expressive quality to more bespoke tasks, such as *pandas* [3] for many statistical tasks required by randomness tests, and *Click* [4] for designing a command-line interface (3).

The rest of this report details the implementation of this suite, titled *coinflip*. It is named in relation to the fact that truly random binary sequences would be equivalent to consecutive (fair) coin flips.

2 Randomness tests

The focus of coinflip was to implement the statistical tests recommended by NIST's own paper [1] that underlies their statistical test suite *sts*, as most tests documented come with walked-through examples to make implementing them easier. The paper will be referred to as it's internal publication code *SP800-22*.

All the tests are available in the `coinflip.randtests` subpackage (structure described in section 2.3.4), where *randtests* stands for *randomness tests*.

2.1 Generalising solutions

NIST's specified randomness tests were detailed with taking binary sequences as input—where binary implies *any* 2 distinct values. However practically they were implemented to only accept sequences with integer values of `0` and `1`.

This lead to the opportunity of implementing tests that accepted *any* binary sequence. This would be particularly useful in eliminating frustrating situations for users, where passed input that *should* work does not, and remove the need for prior conversions.

2.1.1 Sequence representation

The Python package *pandas* [3] was employed for its `Series` data structure, which represent one-dimensional arrays. `Series` objects contain numerous useful methods for data science tasks, made performative due to internal memory optimisations of the underlying data.

As `Series` can hold basically any kind of continuous data, the main consideration of tests accepting any binary sequences was that the sequence was indeed binary (contained 2 and only 2 distinct values). This was achieved by use of `nunique()` method, which returns the count of distinct values.

```
def example_randtest(series, ...):
    if series.nunique() != 2:
        raise NonBinarySequenceError()
    ...
```

Listing 1: If distinct values count of the sequence is not 2 (i.e. is not binary), an error is raised.

To further reduce user friction, we can also make the tests accept any data *type* that represents a binary sequence. In Python 3, this specifically means the object passed as the sequence implements an *Iterable* protocol, i.e. object can be iterated upon.

The pandas `Series` can be initialised with such objects, so we can convert the passed sequence to a `Series`. Conversely, this lets the randomness tests to internally work with a standard type (the `Series`), whilst accepting many possible sequence representations (e.g.: lists, iterators, generators, ndarrays) as long as they implement the idiomatic Iterable interface.

```
def example_randtest_public(sequence, ...):
    if isinstance(sequence, pd.Series):
        series = sequence
    else:
        series = pd.Series(sequence)
        ...
```

Listing 2: If a sequence is not a `Series`, we initialise the `sequence` as one.

2.1.2 Revising test logic

As mentioned previously, NIST created their own randomness test suite *sts* to implement the SP800-22 randomness tests. In reviewing the code of other SP800-22 implementations, many code idioms of sts persist. Infact, the SP800-22 paper explicitly specifies how to actually implement their tests. But in creating *generalised* randomness tests to binary sequences, many of the implementation details need reworking.

1. The zeros and ones of the input sequence (ϵ) are converted to values of 1 and +1 and are added together to produce $S_n = X_1 + X_2 + ... + X_n$, where $X_i = 2\epsilon_i - 1$.

   e.g. if $\epsilon = 1011010101$, then $n = 10$ and $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$.

Figure 1: The first step SP800-22 provides in its walkthrough of how the *Monobits* test works.

For example, in the *Monobits* test specification in SP800-22 (pp. 24-25), there is a concept of proportion between the two values in the inputted sequence (`0` and `1`). SP800-22 states that to find this proportion a variable S_n needs to be calculated, where the `0` values should be replaced with the value `-1`, and the new sequence of `-1` and `1` values should all be summed up together.

```
def monobits(bits):
    prop = 0
    for bit in bits:
        if bit == 1:
            prop += 1
        else:
            prop -= 1
            ...
```

Listing 3: An interpretation of the method specified in figure 1 to find the proportion of `0` and `1` in a sequence.

However this method will not work with sequences that do not contain `0` and `1` values. But on inspection, the resulting value S_n is simply the difference in the count of one value with the other value. This means the test can find the counts of the values and then find the difference afterwards.

Idioms like in the *Monobits* test specification persist throughout SP800-22. In writing generalised randomness tests, they have had to be replaced with new processes that never relate to the *values* of the inputted sequence.

```python
def monobits(series):
    counts = series.value_counts()
    diff = abs(counts.iloc[0] - counts.iloc[1])
    ...
```

Listing 4: The general solution used to find the proportion of the two values in a binary sequence. The `value_counts()` method in pandas `Series` tallies the counts of every value into a new `Series`.

2.1.3 Candidates

SP800-22 describes and implements some randomness tests which specifically depend on the actual values of the sequence, where the values were only assumed to be `0` and `1`.

To meet the goal of implementing these tests to work with sequences that do not contain `0` and `1`, a new concept had to be created called *candidates*. A candidate was a value present in the sequence, which took place of the `1` value referred to in SP800-22, and subsequently the non-candidate could be represent SP800-22's `0` value.

```python
def discrete_fourier_transform(series, candidate):
    peaks = candidate
    trough = next(value for value in series.unique() if value != candidate)

    oscillations = series.map({peaks: 1, trough: -1})
    fourier = fft(oscillations)
    ...
```

Listing 5: The *Discrete Fourier Transform* test treats the candidate as the `peak` of a wave and the other value as the `trough`, to represent the `oscillations` of a sequence for a subsequent Fourier transformation.

To prevent confronting users with the necessary but confusing candidate concept, the `candidate` argument to the implemented randomness tests was made an optional keyword argument. If it wasn't specified, a deterministic method `infer_candidate()` was used to pick a value as a candidate. This method is always used for inference in the tests, so the candidate value is always consistent for a given sequence.

```
def infer_candidate(unique_values):
    try:
        candidate = max(unique_values)
    except TypeError:
        candidate = unique_values[0]

    return candidate
...
def example_randtest(series, candidate=None ...):
    if candidate is None:
        unique_values = series.unique()
        candidate = infer_candidate(unique_values)
        ...
```

Listing 6: The `infer_candidate()` method body, and an example of a randomness test calling it when no `candidate` is passed. Using the `unique()` method from pandas `Series` returns a list of distinct values.

2.2 Randomness test output

A user with no knowledge of the randomness tests will find the p-value itself completely useless in understanding the properties of the RNG output being tested; potentially they believe their output "passed" the tests in that its p-value was over a threshold such as of 0.05 (NIST recommends 0.01), but be none-the-wiser about issues with how they've used the tests.

In implementing a given randomness test, the specific task of finding the correct p-value was only a first step. Care is given to explain to the user how the test works, what input the test can accept, and what NIST recommends on the test's use.

2.2.1 Results

All implemented tests return an object that contains the test-related variables, including the statistic and p-value. This way, users can access interesting variables at any time after running the test, as opposed to only seeing them during the tests execution via logging.

To create the classes of these objects, the code generator *dataclass* from Python's standard library is used. The `@dataclass` decorator offer many conveniences to create classes which primarily are data containers, namely in automating `__init__` methods to initialise with the type-hinted variables defined in the class variable namespace.

```
from dataclasses import dataclass

@dataclass
class TestResult:
    statistic: Union[int, float]
    p: float
    ...
...
def monobits(series):
    n = len(series)
    counts = series.value_counts()
    diff = abs(counts.iloc[0] - counts.iloc[1])
    normdiff = diff / sqrt(n)
    p = erfc(normdiff / sqrt(2))

    return MonobitsTestResult(normdiff, p, n, diff, counts)

@dataclass
class MonobitsTestResult(TestResult):
    counts: pd.Series
    n: int
    diff: int
    ...
```

Listing 7: The parent `TestResult` class, and the *Monobits* test method returning the `MonobitsTestResult` that hold its test constants.

As the returned result is of a class specific to the respective randomness test, a custom `__str__` method can be defined, which is the idiomatic Python method for string representation of objects. This allows the result to be formatted in a human-readable and descriptive fashion when the result object is printed.

```
>> from coinflip.randtests import monobits
>> result = monobits([1, 0, 1, 1, 0, 1, 0, 1, 0, 1])
>> result
MonobitsTestResult(statistic=0.6324555320336759, p=0.5270892568655381,
counts={1: 6, 0: 4}, n=10, diff=2)
>> result.p
0.5270892568655381
>> print(result)
normalised diff  0.632
p-value          0.527

value      count
~~~~~~~  ~~~~~~~
1              6
0              4
```

Listing 8: The `monobits` method returning a `MonobitsTestResult`, and examples of how it could be interacted with.

For the `MonobitsTestResult` specifically, integration of the chart rendering library *matplotlib* [5] was prototyped. The methods `plot_counts()` and `plot_reference_dist()` generate chart objects in relation to the variables of the specific test run, and users of notebooks with matplotlib support (such as Jupyter Notebook [6]) will have the charts automatically render on screen when the respective plot methods are called.

A report generating method was also prototyped in `report()`, which generates HTML markup that explains to the user how the test worked, relating to the variables of the specific test run. The charts are also included in the markup as embedded SVG images. An example of the output can be found in appendix A.

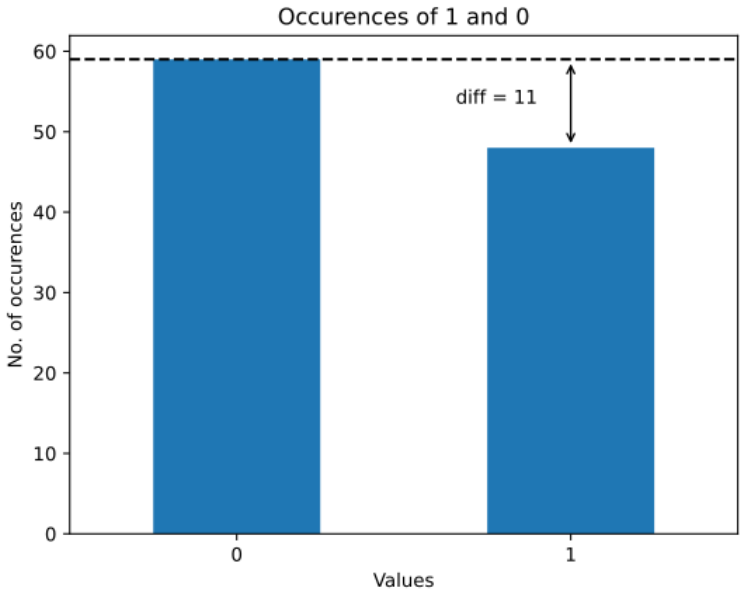


Figure 2: Example output of the `plot_counts()` method in `MonobitsTestResult`.

```
from base64 import b64encode
from io import BytesIO
from matplotlib.figure import Figure
...
@dataclass
class TestResult:
    ...
    @classmethod
    def _markup(cls, item):
        ...
        elif isinstance(item, Figure):
            base64 = fig2base64(item)

            return f"<img src=' {base64}' />"

def fig2base64(fig):
    binary = BytesIO()
    fig.savefig(binary, format="svg")

    binary.seek(0)
    base64_bstr = b64encode(binary.read())
    base64_str = base64_bstr.decode("utf-8")

    return base64_str
```

Listing 9: The class method `_markup()`, which returns the appropriate HTML markup of atomic items, depending on the item's type. For matplotlib `Figure` chart objects, the `fig2base64()` method converts the item to a binary representation of the SVG-equivalent markup of the chart, and then encodes it in a base64 string. This string can be directly embedded in HTML documents and will render appropriately.

A vision for coinflip would be to have these respective chart and report methods for every randomness tests' result object. The report markups could be combined into a singular report on the same RNG output, being a learning opportunity to those not familiar with how all the randomness tests work—and even aid those not familiar with hypothesis testing in statistics generally.

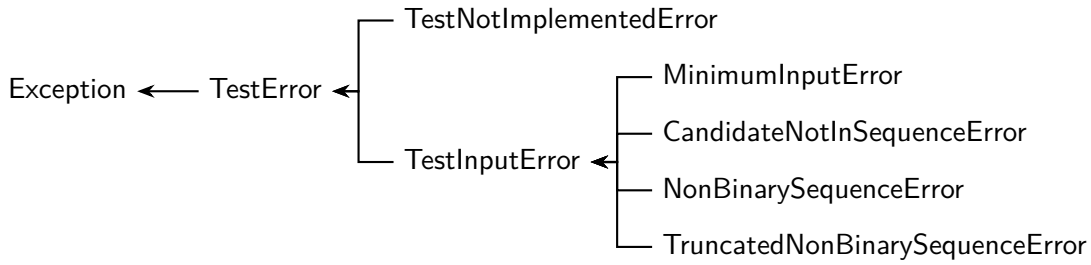


Figure 3: Exception tree for the `coinflip.randtests` subpackage.

2.2.2 Exceptions

The SP800-22 randomness tests can often accept input but fail to process it, so exceptions are raised to halt the test and communicate to the user what went wrong.

Custom exceptions were made for specific scenarios, to allow for control over the messaging and allow other programs only catch the particular scenarios they want to (via the `except` clause). A base error `TestError` parents all errors in the randomness tests, allowing for `except` clauses to cover all test-related errors.

All exceptions implement a `__str__` method. As exceptions are objects, they are initialised, and so can be passed values via the `__init__` too, which allows error messages can specify the parameters involved.

```
class MinimumInputError(TestInputError):
    def __init__(self, n, min_input):
        self.n = n
        self.min_input = min_input

    def __str__(self):
        return (
            f"Sequence length {self.n} is below "
            f"required minimum of {self.min_input}"
        )
...
def example_randtest(series, ...):
    min_input = 128
    n = len(series)
    if n < min_input:
        raise MinimumInputError(n, min_input)
```

Listing 10: As the `MinimumInputError` error relates to `n` and `min_input`, the error can be initialised with these values to give a specific message to the user.

2.2.3 Warnings

SP800-22 recommends a minimum input size for all tests, which presented an opportunity to programactically warn users when their sequence is below the recommended input `rec_input`.

```
rec_input = 1024

def example_randtest(series, ...):
    n = len(series)
    if n < rec_input:
        warn(
            f"Sequence length {n} below NIST "
            "recommended minimum of {rec_input}"
        )
```

Listing 11: Warn user when sequence length `n` is below recommended input `rec_input`.

SP800-22 suggests more complex recommendations too, where parameters of tests should fit certain criteria, relative to the parameters themselves.

The method `check_recommendations()` was created to be passed a dictionary of string representations of the recommendations (the keys), and whether the recommendations passed or not (the values). After the tests decided set their parameters, tests can call this method with a dictionary that is evaluated on-the-fly, and the appropriate warnings are made.

```
def check_recommendations(recommendations):
    failures = [expr for expr, success in recommendations.items() if not success]

    msg = "NIST recommendations not met:\n"
    msg += "\n".join([f"\t {expr}" for expr in failures])
    warn(msg)
...
def frequency_within_block(series, ..., blocksize=8):
    n = len(series)
    nblocks = n // blocksize

    check_recommendations({
        "blocksize  20": blocksize >= 20,
        "blocksize > 0.01 * n": blocksize > 0.01 * n,
        "nblocks < 100": nblocks < 100,
    })
    ...
```

Listing 12: Simplified `check_recommendations()` method, and an example of the *Frequency within Block* test using it.

2.3 Code health

Great care was taken to abstract the mechanisms of the randomness tests, making the literal code of the tests' methods to only be concerned with the top-level logic of the test, and not clunky implementation details.

This separation of concerns kept navigation of the code repository sensible as features were added. This was especially helpful when bugs appeared, as particular aspects of coinflip were isolated for easier review and fixing.

2.3.1 Refactoring test methods

Common patterns in all the randomness tests emerged whilst developing the test suite:

1. Iterable-to-`Series` conversion logic (listing 2)
2. Checking the passed sequence contains 2 distinct values (listing 1)
3. Checking the passed sequence is above test-specific absolute minimum input (listing 10)
4. Warning the passed sequence is below test-specific recommended input (listing 11)

A decorator factory `@randtest()` was created to refactor all of these processes. Patterns 1 and 2 are independent of user input, but 3 and 4 depend on the specific test, and so `@randtest()` can be called with `min_input` and `rec_input` arguments to specify the corresponding behaviour.

Another common process was the inferring of candidate values if not specified (listing 6), which could also be refactored via a decorator, and so `@elected` was made.

```python
def randtest(min_input=2, rec_input=2):
    def decorator(func):
        def wrapper(sequence, *args, **kwargs):
            if isinstance(sequence, pd.Series):
                series = sequence
            else:
                series = pd.Series(sequence)

            if series.nunique() != 2:
                raise NonBinarySequenceError()

            n = len(series)
            if n < min_input:
                raise MinimumInputError(n, min_input)
            if n < rec_input:
                warn(
                    f"Sequence length {n} below NIST "
                    "recommended minimum of {rec_input}"
                )

            result = func(series, *args, **kwargs)

            return result

        return wrapper

    return decorator
...
@randtest(min_input=4, rec_input=387840)
def maurers_universal(series, ...):
    ...
```

Listing 13: The decorator factory `@randtest()`, and an example of its usage in the *Maurer's Universal* test.

```
def elected(func):
    def wrapper(series, *args, candidate=None, **kwargs):
        values = series.unique()
        if candidate is None:
            candidate = infer_candidate(values)
        else:
            if candidate not in values:
                raise CandidateNotInSequenceError()

        result = func(series, *args, candidate=candidate, **kwargs)

        return result

    return wrapper
...
@randtest(min_input=8, rec_input=100)
@elected
def frequency_within_block(series, candidate, ...):
    ...
```

Listing 14: The `@elected` decorator, and an example of its usage in the *Frequency within Block* test.

2.3.2 Custom containers

SP800-22 specified default parameters for some randomness tests, given the sequence length `n` was in a given range. NIST's implementations and others would create if-else blocks to capture the correct `n` and specify the default parameters.

Given the number of `n` ranges, `if` and `else` statements, and variable assignments (i.e. `blocksize = 8`), the code involved was confusing to read and detracted from implementing and maintaining the actual test logic.

As defaults were essentially a dictionary of minimum `n` range (the keys) to default parameters (the values), the defaults could be declared as a Python `dict`. By subclassing `dict` to make passed keys round down to the nearest minimum to create a `FloorDict`, the defaults dictionary `n_defaults` could be accessed with any `n` value, and return the appropriate default parameters.

```
class FloorDict(dict):
    def __missing__(self, key):
        prevkey = None
        for realkey, value in self.items():
            if key < realkey:
                if prevkey is None:
                    raise KeyError()
                return super().__getitem__(prevkey)
            prevkey = realkey
        else:
            return super().__getitem__(prevkey)
...
class Params(NamedTuple):
    blocksize: int
    init_nblocks: int

n_defaults = FloorDict({
    387840: Params(6, 640),
    904960: Params(7, 1280),
    ...
    1059061760: Params(16, 655360),
})

def maurers_universal(series, ...):
    n = len(series)
    blocksize, init_nblocks = n_defaults[n]
    ...
```

Listing 15: The `FloorDict` floors non-existent keys to the nearest real key. Use of `FloorDict` in the *Maurer's Universal* test. Declaring `n_defaults` in such a way allows an idiomatic approach to defaulting parameters.

In implementing the *Longest Run within Block* test, a similar problem occurred.

SP800-22 describes bins for `maxlen`, the length of the longest run (uninterrupted sequence of the same value) within a block. When `maxlen` was below the smallest bin interval, the count for the smallest interval bin was incremented (`maxlen` is rounded up). Subsequently, when `maxlen` above the largest bin interval, the count for the largest bin was incremented (`maxlen` is rounded down).

Like with implementing default parameters, using functional methods (i.e. `if` and `else` statements) made implementing the test confusing and would lead to less maintainable code. Conversely, the problem of incrementing the correct bin count could be defined in terms of a `dict` of intervals (the keys) with initial counts of `0` (the values).

Incrementing the correct bin count when the passed key was out of range was handled by a `RoundingDict`, which wrapped the `dict` getter and setter methods to round these keys. A custom constructor can pass the desired range of intervals and initialise them with values of `0`, which was implemented in the `Bins` container that subclasses `RoundingDict`.

```
class RoundingDict(dict):
    def _roundkey(self, key):
        realkeys = list(self.keys())
        minkey = realkeys[0]
        midkeys = realkeys[1:-1]
        maxkey = realkeys[-1]

        if key <= minkey:
            return minkey
        elif key in midkeys:
            return key
        elif key >= maxkey:
            return maxkey
        else:
            raise KeyError()

    def __setitem__(self, key, value):
        realkey = self._roundkey(key)
        super().__setitem__(realkey, value)

    def __getitem__(self, key):
        realkey = self._roundkey(key)
        return super().__getitem__(realkey)

class Bins(RoundingDict):
    def __init__(self, intervals):
        empty_bins = {interval: 0 for interval in intervals}
        super().__init__(empty_bins)
```

Listing 16: The `RoundingDict` and `Bins` implementations.

```python
from coinflip.randtests._collections import Bins
...
def longest_runs(series, ...):
    ..., intervals = n_defaults[n]     # intervals = [1, 2, 3, 4]
    maxlen_bins = Bins(intervals)      # maxlen_bins = {1:0, 2:0, 3:0, 4:0}
    ...
    for block in blocks(series, ...):
        maxlen = ...                   # maxlen = 6
        maxlen_bins[maxlen] += 1       # maxlen_bins[6] += 1
        ...                            # maxlen_bins = {1:0, 2:0, 3:0, 4:1}
```

Listing 17: Use of `Bins` to in the *Longest Runs within Block* test to succinctly increment the correct bin count with *any* `maxlen` value.

2.3.3 Removing magic numbers

Whilst SP800-22 does explain test-logic, it sometimes only elicits the general idea of the test and not demonstrate the mathematical theories that underly constants used in calculating test variables. This is not a problem for simply implementing the tests, but is insufficient when meeting the goal of *explaining* results (section 2.2.1), where the theories should be expressed programactically themselves, and respective formulas should be used to calculate the respective test variables.

SP800-22 does always provide references for where constants were taken from, which means further investigation *should* lead to any formulas used. However, these references are not necessarily freely accessible online, meaning further exploration has to be done.

For example, in the *Longest Run within Block* test, SP800-22 specifies probabilities of the longest consecutive run in a block to be of a certain length `maxlen`, given the size of a block `blocksize` (further explanation in section 2.3.2).

The citation given for these values was *Random Walk in Random and Non-Random Environments* [7], which was practically inaccessible, and so reproducing the values of table 1 was a goal set in the spirit of exploring how the test works.

This eventually lead to reproducing the distributions of the `maxlen` in *every* permutation of a binary sequence in a notebook session (see appendix B). The number of each `maxlen` occurrences could be divided by the total number of permutations to calculate the same probabilities in table 1.

Table 1: Probabilities provided by SP800-22 for the *Longest Run within Block* test when `maxlen` of a block is in a certain interval, given a `blocksize` of 8.

`maxlen`	Probability
≤ 1	0.2148
2	0.3672
3	0.2305
≥ 4	0.1875

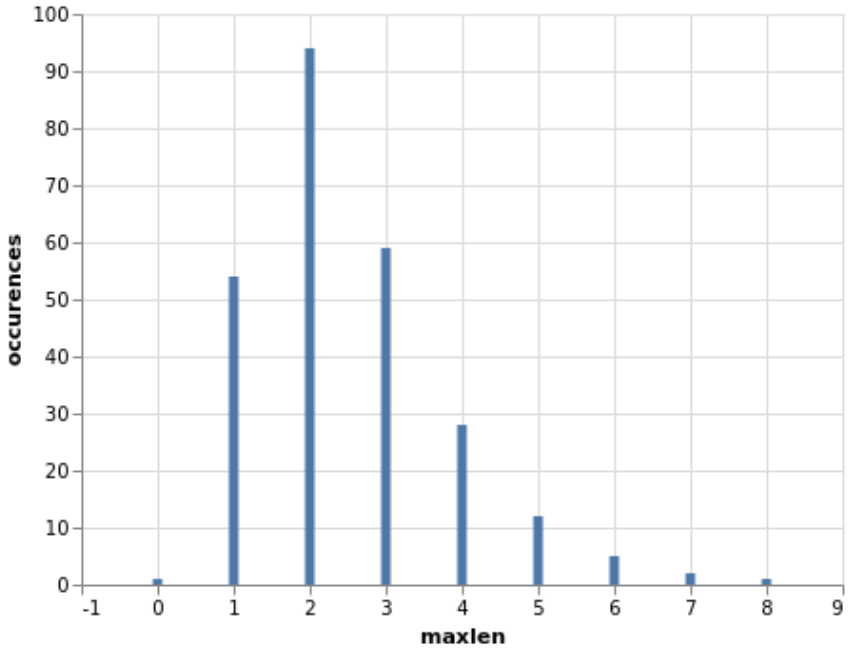


Figure 4: Distribution of `maxlen` in every permutation of a binary sequence with a `blocksize` of 8.

A correct understanding of how these probabilities were calculated allowed the specificity in searching the web, leading to a particular explanation [8] which provided the necessary theory and a formula to calculate such probabilities.

Note this has yet to be implemented, due to the further exploration needed in SP800-22 specifications for certain `maxlen` intervals per `blocksize` (e.g. find out why they use the $\leq$1, 2, 3, and $\geq$4 `maxlen` bins when `blocksize` is 8).

2.3.4 Package structure

The randomness tests are all defined in the `coinflip.randtests` subpackage. Conversely all related code is in the `src/coinflip/randtests/` folder—a full directory tree is available in appendix C.

Randomness tests of a similar nature, such as the *Runs* and *Longest Run within Block* tests, had their methods stored in the same submodule (e.g. `runs.py`). This was generally useful for organisation, but also some specific helper methods and classes could be encapsulated in these modules (e.g. the `asruns()` method in `runs.py`).

All other helpers are in the submodules that start with an underscore `_`, where related modules were kept in the same submodule. These files represent the common library for the randomness tests.

The `coinflip.randtests` subpackage imports all the randomness tests to it's top-level namespace, which means importing it will not expose any submodules, only the test methods. This definitive assortment of randomness tests is utilised in the coinflip CLI (section 3.2), and also establishes a public API to access the tests for notebook users and developers.

```
from coinflip.randtests.fourier import discrete_fourier_transform
from coinflip.randtests.frequency import frequency_within_block
from coinflip.randtests.frequency import monobits
from coinflip.randtests.matrix import binary_matrix_rank
from coinflip.randtests.runs import longest_runs
from coinflip.randtests.runs import runs
from coinflip.randtests.template import non_overlapping_template_matching
from coinflip.randtests.template import overlapping_template_matching
from coinflip.randtests.universal import maurers_universal

__all__ = [
    "monobits",
    "frequency_within_block",
    "runs",
    "longest_runs",
    "binary_matrix_rank",
    "discrete_fourier_transform",
    "non_overlapping_template_matching",
    "overlapping_template_matching",
    "maurers_universal",
]
```

Listing 18: The contents of `__init__.py` in the `coinflip.randtests` subpackage.

Table 2: Responsibilities of submodules in `src/coinflip/randtests/`.

File	Responsibilities
`fourier.py`	*Discrete Fourier Transform* test
`frequency.py`	*Monobits* and *Frequency within Block* test
`matrix.py`	*Binary Matrix Rank* test
`runs.py`	*Runs* and *Longest Run within Block* tests
`template.py`	*(Non-)Overlapping Template Matching* tests
`universal.py`	*Maurer's Universal* test
`_collections.py`	Custom contains (2.3.2)
`_decorators.py`	Test method refactoring (2.3.1)
`_exceptions.py`	Base and common exceptions (2.2.2)
`_pprint.py`	Pretty printing sequences
`_result.py`	`TestResult` (2.2.1)
`_tabulate.py`	Table string formatting
`_testutils.py`	Common methods for test logic
`__init__.py`	Import hacking of test methods

3 Command-line interface

The Python package Click [4] is used to define the commands of coinflip. Methods represent commands, with decorators defining the arguments and options of the command, and docstrings defining help text.

```
from click import group, argument, option
...
@group()
def main():
    pass
...
@main.command()
@argument("store", ...)
@option("-t", "--test", ...)
def run(store, test):
    """Run randomness tests on data in STORE.

    Results of the tests run are saved in STORE, which can be compiled into
    a rich document via the report command.
    """
    ...
```

Listing 19: Simplified `main()` method which groups other commands together in the CLI, and simplified `run()` method which represents the `coinflip run` command.

All command methods are kept in the `cli.py` module, but the store (section 3.1) and test running (section 3.2) functionality is assigned to the `store.py` and `tests_runner.py` submodules respectively—a full directory tree is available in appendix C. Even though this functionality is for the exclusive use of `cli.py`, it was found easier to maintain the store and test running logic when they were isolated into their own files.

Beyond declaring the CLI commands themselves, `cli.py` is simply responsible for all subsequent communication to the user. This includes handling the warnings and errors of the storing and test running methods, so that human-readable messages are printed—not the verbose and noisy tracebacks of the Python interpreter.

```
$ coinflip -h
Usage: coinflip [OPTIONS] COMMAND [ARGS]...

  Randomness tests for RNG output.

  Output of random number generators can be parsed and serialised into a
  test-ready format via the load command. The data is saved in a folder,
  which coinflip refers to as a "store". This store is located in the local
  data directory, but can be easily accessed via the store's name in coinflip
  commands.

  Randomness tests can then be ran over the store's data via the run
  command. Rich documents explaining the test results can be produced via
  the report command.

Options:
  -h, --help  Show this message and exit.

Commands:
  cat          Print contents of data in STORE.
  example-run  Run randomness tests on example data.
  load         Loads DATA into a store.
  local-run    Run randomness tests on DATA directly.
  ls           List all stores.
  report       Generate html report from test results in STORE.
  rm           Delete STORE.
  rm-all       Delete all stores.
  run          Run randomness tests on data in STORE.
```

Listing 20: Automatically generated description of the `coinflip` command-line interface.

```
$ coinflip run -h
Usage: coinflip run [OPTIONS] STORE

  Run randomness tests on data in STORE.

  Results of the tests run are saved in STORE, which can be compiled into a
  rich document via the report command.

Options:
  -t, --test <test>  Specify single test to run on data.
  -h, --help         Show this message and exit.
```

Listing 21: Description of the `coinflip run` command.

```
import warnings
from colorama import Fore, Style
...
warn_txt = Fore.YELLOW + "WARN" + Fore.RESET

def formatwarning(msg, *args, **kwargs):
    return Style.DIM + f"{warn_txt} {msg}\n" + Style.RESET_ALL

warnings.formatwarning = formatwarning
```

Listing 22: Monkey patching of Python's `warning` module to use a custom `formatwarning()` method, which pretty prints warnings.

```
from click import echo
...
err_txt = Fore.RED + "ERR!" + Fore.RESET

def echo_err(e: Exception):
    line = f"{err_txt} {e}"
    echo(line, err=True)
```

Listing 23: The `echo_err()` method, which pretty prints exceptions.

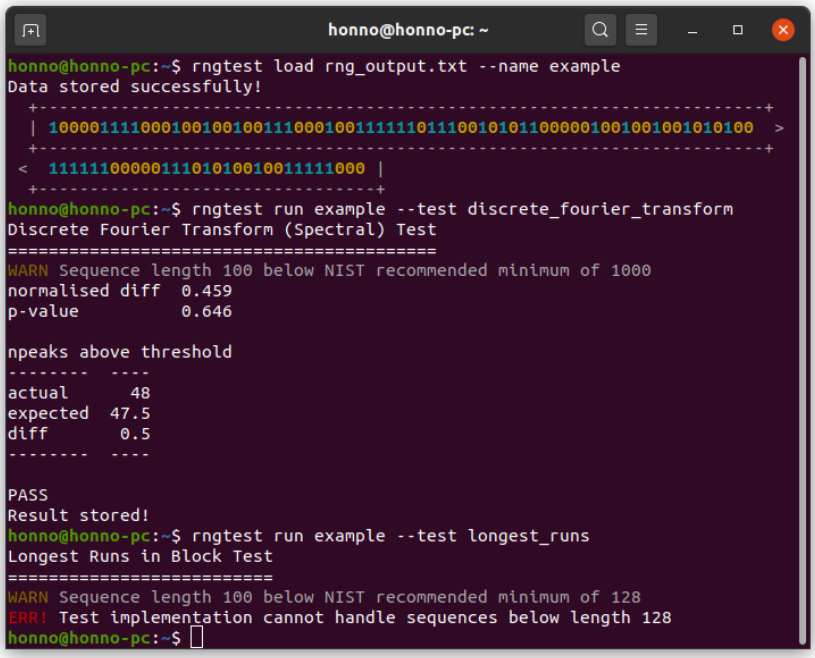


Figure 5: Example terminal use of the coinflip CLI.

3.1 Stores

A *store* is simply an abstraction of the concept that pairs of binary sequences and their test results should be kept together and made distinct from other such pairs.

Operating systems in the Windows, Mac and Linux ecosystems all have directories meant for the use of user data in third-party applications. In the CLI, stores represent folders in these user data directories.

The Python package *appdirs* [9] provides a cross-platform abstraction to operating system directories, including the user data directory. Internally the user data directory is referred to as `data_dir`, which is a Python `Path` object that represents to app folder's location in the filesystem.

One reason for this store abstraction is to enforce a tidy filesystem for the user.

```
$ coinflip ls
$ coinflip load binary_sequence.txt
Store name to be encoded as store_20200721T093641Z
Data stored successfully!
...
$ coinflip ls
store_20200721T093641Z
```

Listing 24: Using the `coinflip ls` command, there are initially no stores. Loading RNG output via the `coinflip load` command, the subsequent call of `coinflip ls` shows that there is now a store of the aforementioned RNG output.

```
$ ls $HOME/.local/share/coinflip
$ coinflip load binary_sequence.txt
Store name to be encoded as store_20200721T093641Z
Data stored successfully!
...
$ ls $HOME/.local/share/coinflip
store_20200721T093641Z
```

Listing 25: Using GNU's `ls` command on coinflip's user data directory on Ubuntu, there are initially no folders representing stores. Loading RNG output via the `coinflip load` command, the subsequent call of `ls` shows that there is now a folder representing a store.

```
from appdirs import AppDirs
...
dirs = AppDirs(appname="coinflip", appauthor="MatthewBarber")
data_dir = Path(dirs.user_data_dir)

try:
    Path.mkdir(data_dir, parents=True)
except FileExistsError:
    pass
```

Listing 26: The appdirs package being used to identify the user data directory, and create a folder for the coinflip app if it does not already exist.

The results being stored somewhere is very helpful, because whilst the CLI may output all the necessary info, it can be tedious for users to store such output if they're not familiar with workflows to store standard output (e.g. `coinflip run my_store > results.txt` in bash).

In the other suites, randomness tests are run directly on a file containing RNG output, which became confusing when multiple invocations of the tests are run—the suites can only guess where to place the results are (e.g. a `log.txt` file).

An abstraction to access the results enforces an app-specific user data directory as the canonical location of results, which can also be conveniently accessible via the coinflip CLI itself.

Another important reason for store abstractions is the step-by-step process of running tests it allows. Often a source of frustration is ensuring the actual RNG output is being correctly handled by the suite. Having the command `coinflip load` isolates this process, so new users can get to grips with how their RNG output should be formatted—if their first attempt fails, the only error messages given are specific to the process of loading the RNG output.

```
$ coinflip load -h
Usage: coinflip load [OPTIONS] DATA

  Loads DATA into a store.

  DATA is a newline-delimited text file which contains output of a random
  number generator. The contents are parsed, serialised and saved in local
  data.

  The stored data can then be applied the randomness tests via the run
  command, where the results of which are also saved.

Options:
  -n, --name TEXT      Specify name of the store.
  -d, --dtype <dtype>  Specify data type of the data.
  -o, --overwrite      Overwrite existing store with same name.
  -h, --help           Show this message and exit.
```

Listing 27: Description of the `coinflip load` command.

Someone familiar with a tool such as coinflip can come to prefer a single step to run a test on RNG output and output the results, and so the `coinflip local-run` command was made to combine steps.

```
$ coinflip local-run -h
Usage: coinflip local-run [OPTIONS] DATA

  Run randomness tests on DATA directly.

Options:
  -d, --dtype <dtype>  Specify data type of the data.
  -t, --test <test>    Specify single test to run on data.
  -h, --help           Show this message and exit.
```

Listing 28: Description of the `coinflip local-run` command.

The store abstraction would also useful if the HTML report concept (section 2.2.1) came to fruition. A command such as `coinflip report` could generate the final report file (i.e. `report.html`) and automatically open it in the users preferred browser (Python's *webbrowser* module in its standard library can handle this in all platforms).

3.1.1 Loading in RNG output

A user "loading" their RNG output is infact initialising store with that RNG output serialised as a pandas `Series`.

The RNG output is first parsed. pandas provides a `read_csv()` method that can smartly assess the contents and load it into a pandas `DataFrame`. The method assumes columnar data, but as the RNG output should be a *single continuous* sequence, an error is raised if multiple columns are parsed. Like in listing 1, the resulting `Series` is checked to be a *binary* sequence.

If the data is successfully parsed, a store is then initialised. If a name is provided by the `--name` option for the `coinflip load` command, that is sanitised and used as the store's name; otherwise a name is generated that includes a ISO-8601 [10] formatted timestamp. A check is made if a store of the same name already exists, in which case an error is raised. If however the `--overwrite` flag is used, the old store is overwritten with the new RNG output.

```
def parse_data(data_file):
    df = pd.read_csv(data_file, header=None)

    ncols = len(df.columns)
    if ncols > 1:
        raise MultipleColumnsError(ncols)
    series = df.iloc[:, 0]

    if series.nunique() != 2:
        raise NonBinarySequenceError()

    return series
```

Listing 29: Simplified `parse_data()` method, which reads from file containing RNG output and produces a representative pandas `Series`. Checks are made to ensure the RNG output is a single continuous binary sequence, so as to be run on the randomness tests.

Once the store is initialised—by way of a folder being created in the user data directory—the parsed data is serialised via the `pickle` module in Python. The module conveniently saves the `Series` representation of the RNG output as the actual bytecode, meaning you can later directly load it as the original `Series`. This is saved as the `series.pickle` file in the store's folder.

```
def init_store(name=None, overwrite=False):
    if name:
        store_name = slugify(name)

    else:
        timestamp = datetime.now()
        iso8601 = timestamp.strftime("%Y%m%dT%H%M%SZ")
        store_name = f"store_{iso8601}"

    store_path = data_dir / store_name
    try:
        Path.mkdir(store_path, parents=True)
    except FileExistsError:
        if overwrite:
            rm_tree(store_path)
            Path.mkdir(store_path)
        else:
            raise StoreExistsError(store_name)

    return store_name, store_path
```

Listing 30: Simplified `init_store()` method which initialises a store. A store name can be supplied via the `name` argument, or generated. If a name conflicts with an existing store, the `overwrite` argument decides whether to overwrite it.

```
import pickle
...
def store_data(data_file, name=None, overwrite=False):
    series = parse_data(data_file, dtype_str)

    store_name, store_path = init_store(name=name, overwrite=overwrite)

    data_path = store_path / "series.pickle"
    pickle.dump(series, open(data_path, "wb"))
```

Listing 31: Simplified `store_data()` method which makes use of `parse_data()` and `init_store()` to streamline the process of loading user data in a store in the `coinflip load` command.

```python
from coinflip.randtests._exceptions import NonBinarySequenceError
from coinflip.store import store_data, DataParsingError, StoreError
...
@main.command()
@argument("data", ...)
@option("-n", "--name", ...)
@option("-o", "--overwrite", ...)
def load(data, name, overwrite):
    try:
        store_data(data, name=name, overwrite=overwrite)
    except (DataParsingError, NonBinarySequenceError, StoreError) as e:
        echo_err(e)
        exit(1)
```

Listing 32: Simplified `load()` method that represents the `coinflip load` command, calling `store_data()` (listing 31).

3.1.2 Discoverability

Click allows for dynamic autocompletion when enabled by the users shell configuration, meaning beyond hinting fixed inputs such as `--test <test>` (where the possible `<test>` values are always the same), the CLI can also hint at the stores available in the current moment.

Additionally, the CLI will assume the `STORE` argument for commands such as `coinflip run` to be the most recently initialised store. This can smooth the CLI's learning experience for users, and generally be a convenient shortcut.

This functionality achieved by use of a text file `latest_store.txt`, located in the root of the app's user data directory. When `store_data()` (listing 31) succeeds in initialising a store, it will finish by saving the stores name; the `store_name` can then accessed in `find_latest_store()`.

In `cli.py`, command methods can then find the latest store name, and if successful prompt the user whether they wish to use it.

```
def list_stores():
    try:
        for entry in scandir(data_dir):
            if entry.is_dir():
                yield entry.name

    except FileNotFoundError:
        pass
...
def get_stores(ctx, args, incomplete):
    stores = list(list_stores())
    if incomplete is None:
        return stores
    else:
        for name in stores:
            if incomplete in name:
                yield name
...
@main.command()
@argument("store", autocompletion=get_stores, ...)
@option("-t", "--test", ...)
def run(store, test):
    ...
```

Listing 33: The `get_stores()` completion method, which uses the `list_stores()` method to match all possible stores the user's typed `STORE` argument could lead to. An example of its use is in the `run()` method, which represents the `coinflip run` command.

```
$ coinflip load DATA
Store name to be encoded as store_<timestamp>
Data stored successfully!
$ coinflip run
No STORE argument provided
  The most recent STORE to be initialised is 'store_<timestamp>'
  Pass it as the STORE argument? [y/N]:
```

Listing 34: Workflow of loading data into a new store, and calling `coinflip run` without the `STORE` argument.

```python
def store_data(data_file, ...):
    ...
    store_name, ... = init_store(...)
    ...
    latest_store_path = data_dir / "latest_store.txt"
    with open(latest_store_path, "w") as f:
        f.write(store_name)

def find_latest_store() -> str:
    latest_store_path = data_dir / "latest_store.txt"
    try:
        with open(latest_store_path) as f:
            store_name = f.readlines()[0]
            if store_name in list_stores():
                return store_name
    except FileNotFoundError:
        pass

    raise NoLatestStoreRecordedError()
```

Listing 35: The final step in `store_data()`, where `latest_store.txt` is overwritten the initialised store's name. The `find_latest_store()` method then attempts to find the store's name and checks that it is valid, otherwise it raises an error to except.

```
from coinflip.store import find_latest_store, NoLatestStoreRecordedError
...
def infer_store():
    try:
        store = find_latest_store()
    except NoLatestStoreRecordedError:
        ctx = get_current_context()
        ctx.fail("Missing argument 'STORE'.")

    msg = (
        "No STORE argument provided\n"
        f"\tThe most recent STORE to be initialised is '{store}'\n"
        "\tPass it as the STORE argument?"
    )
    if confirm(msg):
        return store
    else:
        exit(0)
...
@main.command()
@argument("store", ...)
@option("-t", "--test", ...)
def run(store, test):
    if not store:
        store = infer_store()
        ...
```

Listing 36: The `infer_store()` method, which is responsible for the `STORE` inference functionality for commands such as `coinflip run`.

3.1.3 Interacting with the store

Tests can be run on stored RNG output via `coinflip run STORE`, where `STORE` is the name of the store to access. Internally, that name is the folder's name, referred to as `store_name`.

The `store_name` can be appended the the `data_dir` to access the store's respective folder. In `coinflip run`, the RNG output is needed to be run on the randomness tests, which is achieved via the `get_data()` method.

```
def get_data(store_name):
    store_path = data_dir / store_name
    if not store_path.exists():
        raise StoreNotFoundError(store_name)

    data_path = store_path / DATA_FNAME
    try:
        with open(data_path, "rb") as f:
            series = pickle.load(f)

        return series

    except FileNotFoundError as e:
        raise DataNotFoundError(store_name)
```

Listing 37: The `get_data()` method which allows access data of a store's RNG output.

Results of the randomness tests are stored in a pickled `dict` via the sibling Python module to `pickle`, `shelve`. These dictionaries are saved in the `results.dbm` file adjacent to the corresponding `series.pickle`.

```
import shelve
...
@contextmanager
def open_results(store_name):
    store_path = data_dir / store_name

    if not store_path.exists():
        raise StoreNotFoundError

    results_path = store_path / "series.pickle"

    with shelve.open(results_path) as results:
        yield results

def store_results(store_name, results_dict):
    with open_results(store_name) as results:
        for randtest_name, result in results_dict.items():
            results[randtest_name] = result
```

Listing 38: The `open_results()` helper context manager, and the `store_results()` method which uses it. Note that `@contextmanager` makes the `open_results()` function into a Python context manager by exposing `results` through a `yield` statement.

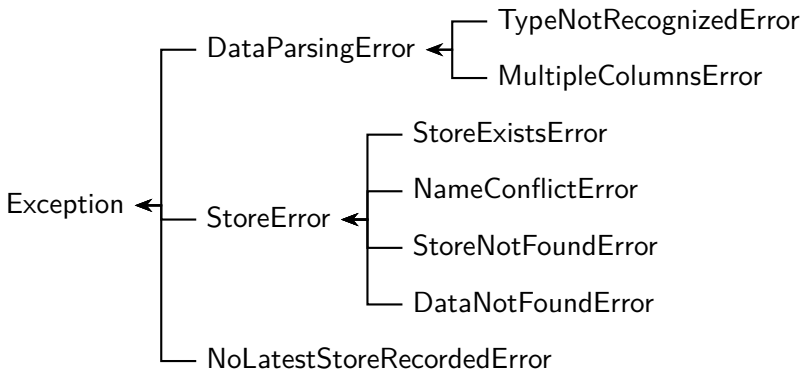


Figure 6: Exception tree of the `coinflip.store` submodule.

3.2 Tests runner

The `tests_runner.py` submodule is responsible running user's data with the randomness tests in the `randtests` subpackage, as well as managing the test-specific CLI messages.

The `run`, `example-run` and `local-run` commands all have two modes of operations—running RNG output on either a single randomness test, or all of them. These cases are handled by the `run_test()` and `run_all_tests()` methods respectively.

In `run_test()`, the `randtests` subpackage is iterated upon it's top-level methods, which solely consist of the randomness tests. These methods are stored in the `__all__` property, and so can be inspected to create a definitive list of all randomness test methods available. Subsequently this list is searched of the `str` names of the tests, to find the corresponding function object of the randomness test that can be executed.

```
from coinflip import randtests
...
def list_tests():
    for randtest_name in randtests.__all__:
        randtest_func = getattr(randtests, randtest_name)

        yield randtest_name, randtest_func
...
def run_test(series, randtest_name, **kwargs):
    for name, func in list_tests():
        if randtest_name == name:
            ...
            result = func(series, **kwargs)
            ...
    else:
        raise TestNotFoundError()
```

Listing 39: The `list_tests()` and `run_test()` methods, which enable users to run randomness tests via a string.

In `run_all_tests()`, all the randomness tests are iterated upon. Instead of halting on test-related errors, they are captured using the `TestError` base exception for randomness tests (section 2.2.2). For every randomness test run, the test's name, result and exception is yielded, to make the method a generator—the result may be `None` if an exception was raised, and conversely the `exception` may be `None` if there were no exceptions raised.

```
def run_all_tests(series):
    ...
    for name, func in list_tests():
        echo_randtest_name(name)

        try:
            result = func(series)
            echo(result)

            yield name, result, None

        except TestError as e:
            yield name, None, e

        echo()
        ...
```

Listing 40: The `run_all_tests()` method, which handles running all the randomness tests on a series and pretty printing the results.

The exposure of exceptions is used in `cli.py` to pretty print error messages. In the `coinflip run` command particularly, the exposure of the name and result allows the subsequent saving of results into the appropriate store.

```
from coinflip.store import get_data, store_results
from coinflip.tests_runner import run_all_tests
...
@main.command()
@argument("store", ...)
def run(store):
    series = get_data(store)

    results = {}
    for name, result, e in run_all_tests(series):
        if e:
            echo_err(e)
        else:
            results[name] = result

    store_results(store, results)
    echo("Results stored!")
```

Listing 41: Simplified `run()` method that represents the `coinflip run` command. The `run_all_tests()` method (listing 40) handles logging whilst exposing the result, which is saved in the `STORE` specified by the user.

4 Testing

When implementing the randomness tests (2) and command-line interface (3), writing suitable tests to "describe" the intended behaviour of functionality was prioritised first to verify new features would work as intended i.e. test-driven development.

The Python package *pytest* [11] was employed for the ability to write simple tests; any method denoted with `test_` is automatically run by the pytest CLI. pytest can also manage more advanced testing needs, such as the paramtrisation and test-generation hooks discussed in section 4.1.1. Another Python testing package *Hypothesis* [12] features heavily in testing coinflip, and is detailed in the subsequent sub-sections.

The Python testing wrapper tox [13] allows the specification of various testing profiles. In local development, this is useful to run all tests for a quick check that changes haven't broken coinflip. For example, `tox -e py37` provides a quick way to execute a long command on a freshly installed version of coinflip on Python version 3.7.

```
$ pytest -vv tests \
> --cov --cov-report=term-missing \
> --hypothesis-profile=quick  \
> --ignore tests/randtests/implementations/
```

Listing 42: Command that `tox -e py37` wraps.

coinflip also uses continuous integration, employing the Travis CI [14] and AppVeyor [15] services. tox allows specifying all tests and checks that are desired, including the Python versions one wishes to execute them in.

AppVeyor in particular builds and tests coinflip in a Windows environment. As coinflip was developed on Ubuntu, this is helpful in automatically checking that there were no breaking changes in Windows.

One example this was useful was in recognising a problem with the creation of the user data directory (listing 26). Initially `Path.mkdir(data_dir)` was used, which will work fine on popular Unix-based operating systems where user data directories followed a `$HOME/<user_data_dir>/<app_name>/` pattern. However, because Windows has app user data directories specified in `$HOME/<user_data_dir>/<author_name>/<app_name>/`, `Path.mkdir()` could not find a parent `<author_name>` folder and so failed. Subsequently the `parents` keyword argument was added to result in `Path.mkdir(data_dir, parents=True)`.

4.1 Randomness tests

4.1.1 Known test cases

SP800-22 includes worked examples for their recommend randomness tests (for the most part), which was used to help verify the implemented tests. The results of these examples are asserted to the results of compared implementations (which are passed the same parameters).

Additionally, the working out provided in SP800-22 helped in debuging the tests, as the processes of the implemented could be reviewed step-by-step by use of a debugger (i.e. the Python debugger `pdb`) to see where exactly miscalculation occurred.

As the number of pytest tests such as in listing 43 grew, the code became confusing to navigate which frustrated debuging. As all the examples followed a similar pattern, a container `Example` was made to specify the parameters of the test programactically.

1. The zeros and ones of the input sequence (ϵ) are converted to values of 1 and +1 and are added together to produce $S_n = X_1 + X_2 + ... + X_n$, where $X_i = 2\epsilon_i - 1$.

   e.g. if $\epsilon = 1011010101$, then $n = 10$ and $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$.

2. Compute the test statistic $S_{obs} = |S_n| / \sqrt{n}$

   e.g. in this section, $S_{obs} = |2| / \sqrt{10} = 0.632455532$.

3. Compute $p = erfc(S_{obs}/\sqrt{2})$, where $erfc$ is the complementary error function.

   e.g. $p = erfc(0.6324555/\sqrt{2}) = 0.527089$.

Figure 7: Walkthrough of how the *Monobits* test works with examples, provided by SP800-22.

```
from coinflip.randtests import monobits
...
def test_monobits():
    bits = [1, 0, 1, 1, 0, 1, 0, 1, 0, 1]

    result = monobits(bits)

    assert isclose(result.statistic, 0.632455532)
    assert isclose(result.p, 0.527089)
```

Listing 43: Original pytest method to test the *Monobits* randomness test. `bits` is passed as the `sequence` argument to the `monobits` method, and a `result` is captured and compared to the expected statistic and p-value.

```python
class Example(NamedTuple):
    randtest: str
    bits: List[int]
    statistic: Union[int, float]
    p: float
    kwargs: Dict[str, Any] = {}
```

Listing 44: The named tuple `Example`, which contains specification for SP800-22 test examples.

pytest enables parametrisation of test methods by way of the `@pytest.mark.parametrize()` decorator, which enables a list of examples to passed to a general runner of the implemented randomness tests. This meant one definite list `examples` could hold the SP800-22 example specifications, which were more easily navigatable than example-specific test methods.

When working on a test and a specific example was failing, I would want to only test on that example. pytest offers a `-k` option in its command-line tool, which could filter out examples for a specific randomness test, but not a specific example in the case of multiple examples for one randomness test.

Therefore `examples` was declared as a heterogeneous `dict`, where the key(s) would form a specific test name. This can be achieved as pytest can parameterise methods such as `test_randtest_on_example()` dynamically via configuration in `conftest.py` files.

```python
from coinflip import randtests
...
examples = [
    Example(
        randtest="monobits",

        bits=[1, 0, 1, 1, 0, 1, 0, 1, 0, 1],

        statistic=.632455532,
        p=0.527089,
    ),
    ...
    Example(
        randtest="maurers_universal",

        bits=[
            0, 1, 0, 1, 1, 0, 1, 0,
            0, 1, 1, 1, 0, 1, 0, 1,
            0, 1, 1, 1,
        ],
        kwargs={
            "blocksize": 2,
            "init_nblocks": 4,
        },

        statistic=1.1949875,
        p=0.767189,
    )
]

@pytest.mark.parametrize("randtest, bits, statistic, p, kwargs", examples)
def test_randtest_on_example(randtest, bits, statistic, p, kwargs):
    randtest_method = getattr(randtests, randtest)

    result = randtest_method(bits, **kwargs)

    assert isclose(result.statistic, statistic, rel_tol=0.05)
    assert isclose(result.p, p, abs_tol=0.005)
```

Listing 45: Intermediate solution of testing SP800-22 examples, by way of a `Example` list in `examples`, and direct parametrisation on a general method `test_randtest_on_example`.

```
examples = {
    ...
    "binary_matrix_rank": {
        "small": Example(  # "binary_matrix_rank.small"
            ...
        ),
        "large": Example(  # "binary_matrix_rank.large"
            ...
        )
    },
    ...
}
...
def examples_iter(regex = ".*"):
    regexc = re.compile(regex)
    for example_title, example in flatten_examples(examples):
        if regexc.match(example_title):
            yield example
```

Listing 46: How `examples` is declared, and the `examples_iter()` method which interfaces with `examples` given a regex expression. The `flatten_examples()` method generates the concatenated example titles along with the respective `Example`.

```
from .examples import examples_iter
...
def pytest_addoption(parser):
    parser.addoption("-example", default=".*",)

def pytest_generate_tests(metafunc):
    if metafunc.function.__name__ == "test_randtest_on_example":
        title_substr = metafunc.config.getoption("example")
        metafunc.parametrize(
            "randtest, bits, statistic, p, kwargs",
            examples_iter(title_substr)
        )
```

Listing 47: Code in `conftest.py`, which allows use of the `-example <regex>` option to only test the desired examples. `pytest_adoption()` allows customisation of the pytest command-line tool, and `pytest_generate_tests()` allows custom setup hooks for test generation.

4.1.2 Generating test cases

Property-based testing is employed to specify *all* possible input for the randomness tests. This means generating data that meets the input specification at various extremes, which the Hypothesis [12] library provides for.

Possible input for all randomness tests will always include a binary sequences (the RNG output being tested), which can be declared as a Hypothesis *strategy*. Strategies are data generators for property-based testing, and can be combined and manipulated to create new strategies.

`mixedbits()` is such a strategy, devised to generate binary sequences. It wraps a `lists()` strategy, which was specified to generate lists of output from the `integers()` strategy, subsequently specified to return only `0` and `1` numbers (i.e. binary). The resulting strategy is then filtered so only lists with *both* `0` and `1` values are outputted, i.e. mixed bits.

```
from hypothesis import given
from hypothesis import strategies as st
...
def mixedbits():
    binary = st.integers(min_value=0, max_value=1)  # 0 or 1
    bits = st.lists(binary, min_size=2)
    mixedbits = bits.filter(contains_multiple_values)

    return mixedbits
...
@given(mixedbits())
def test_monobits(bits):
    result = randtests.monobits(bits)
    ...
```

Listing 48: Declaration of the `mixedbits()` strategy and an example of it's use on the *Monobits* randomness test. `contains_multiple_values()` is a filter for whether the sequence is multi-valued or not.

This alone can reveal failures in just running supposedly valid data on the implemented randomness tests i.e. smoke testing.

For example, a bug was discovered in *Discrete Fourier Transform* test via the `mixedbits()` strategy. The test only uses an even-lengthed sequence and will truncate any odd bits—this means it can be passed a valid binary sequence, but after truncation the sequence becomes single-valued, and so unexpectedly breaks the fourier transform involved in the test. This unique scenario was only discovered through the generated input data, and now means users are raised a specific `NonBinaryTruncatedSequenceError` which explains the circumstance why the randomness test cannot accept their input.

```
>> from coinflip.randtests import discrete_fourier_transform
>> discrete_fourier_transform([0, 0, 0, 0, 0, 0])
Traceback...
...NonBinarySequenceError: \
Sequence does not contain only 2 distinct values (i.e. binary)
>> discrete_fourier_transform([0, 0, 0, 0, 0, 1])
Traceback...
...NonBinaryTruncatedSequenceError: \
When truncated into an even-length, sequence contains only 1 distinct value
i.e. the sequence was originally binary, but now isn't
```

Listing 49: The `NonBinarySequenceError` and derivative `NonBinaryTruncatedSequenceError` being raised when using the *Discrete Fourier Transform* randomness test.

4.1.3 Comparing other implementations

The results of the randomness tests using generated input can be compared to other implementations. Currently two open source Python programs are used, one by David Johnston [16], and the other by Stuart Gordon Reid [17].

These programs are adapted to interface the same way as my randomness tests, assuming the binary sequences are Python `list` objects of `0` and `1` integers (e.g. `[0,1,0,0,1]`). Adaptors also give the opportunity to make the implementations be more idiomatic to use.

```
from .sp800_22_tests.sp800_22_monobit_test import monobit_test
...
class DJResult(NamedTuple):
    success: bool
    p: float
    unknown: None

def named(randtest):
    def wrapper(bits, *args, **kwargs):
        result = randtest(bits, *args, **kwargs)

        return DJResult(*result)

    return wrapper

@named
def monobits(bits):
    return monobit_test(bits)
```

Listing 50: Decorator `@named` used for the adapted tests of Johnston's program, with an example of its use on the *Monobits* test. It wraps the non-descriptive results of the implementated methods in the named tuple `DJResult`.

The adapted method have non-deterministic or fixed test variables (`kwargs`) in some of the randomness tests, and so an `Implementation` named tuple describe the adapted methods as such. Each randomness test can be mapped to an `Implementation`, to communicate the restrictions in their use.

Additionally, the original methods can accept values but then fail to process them. When the circumstances of these failures are identified, the adapted method can raise an `ImplementationError` when passed parameters lead to such failures. These errors describe when to skip over comparing the adapted method to my randomness tests; programactically allowing for these scenarios to be identified via a `except ImplmenetationError` clause.

```python
from .r4nd0m.SourceCode.RandomnessTests import RandomnessTester
...
def bits_str(randtest):
    def wrapper(bits, *args, **kwargs):
        bits_str = "".join(str(bit) for bit in bits)

        result = randtest(bits_str, *args, **kwargs)

        return result

    return wrapper

@bits_str
def monobits(bits):
    tester = RandomnessTester(None)
    return tester.monobit(bits)
```

Listing 51: Decorator `@bits_str` used for the adapted tests of Reid's program, with an example of its use on the *Monobits* test. It converts the inputted binary sequence into a string of bits, as necessitated by Reid's tests.

The information provided by the `Implementation` wrapper and the raising of `ImplementationError` is sufficient to programactically describe the runtime errors of the implemented tests. This enables running the SP800-22 examples (as described in 4.1.1) on the adapted programs themselves. This is useful for gauging the accuracy of the examples themselves, if both my randomness tests and another implementation fails to have a similar result.

For example, a SP800-22 example for the *Discrete Fourier Transform* test was failing in my own randomness test, however the Johnston and Reid implementations both got nearly the same result as myself. This first suggested an error in the example as opposed to my randomness test, but ultimately led to the insight that we were using the same fourier transform algorithm—fast fourier transform `fft` as part of the SciPy [18] library—which was different to NISTs own fast fourier transform algorithm. Due to time constraints this was not investigated further; it seems likely that with larger sequences the differences will become insignificant however, but it should still necessitate comparing the two algorithms.

```
class Implementation(NamedTuple):
    randtest: Callable
    missingkwargs: List[str] = []
    fixedkwargs: Dict[str, Any] = {}
...
from .sp800_22_tests.sp800_22_non_overlapping_template_matching_test import \
    non_overlapping_template_matching_test as _non_overlapping_template_matching
...
@named
def non_overlapping_template_matching(bits):
    return _non_overlapping_template_matching(bits)
...
dj_testmap = {
    ...
    "non_overlapping_template_matching": Implementation(
        non_overlapping_template_matching,
        missingkwargs=["template"],
        fixedkwargs={"nblocks": 8},
    ),
    ...
}
```

Listing 52: The `Implementation` named tuple and an example of its usage for declaring Johnston's *Non-overlapping Template Matching* test.

Table 3: The p-values of the *Discrete Fourier Transform* test for the same input sequence of `1001010011`.

Implementation	p-value
SP800-22	0.029523
coinflip	0.468159
Johnston	0.468159
Reid	0.468159

```python
@named
def binary_matrix_rank(bits, matrix_dimen):
    nrows, ncols = matrix_dimen
    nblocks = len(bits) // (nrows * ncols)
    if nblocks < 38:
        raise ImplementationError()

    try:
        return _binary_matrix_rank(bits, M=nrows, Q=ncols)
    except (ZeroDivisionError, IndexError) as e:
        raise ImplementationError() from e
```

Listing 53: Adaptor method of Johnston's *Binary Matrix Rank* test. The original method cannot handle `nblocks` below 38, so the passed parameters are checked for that and raise an `ImplementationError`. The original method also sometimes unintentionally raises errors due to non-robust code, so an `ImplementationError` also wraps them.

```python
from pytest import skip
...
def test_randtest_on_example(randtest, bits, statistic, p, kwargs):
    implementation = dj_testmap[randtest]

    if implementation.missingkwargs or implementation.fixedkwargs:
        skip()

    try:
        result = implementation.randtest(bits, **kwargs)
    except ImplementationError:
        skip()

    assert isclose(result.p, p)
```

Listing 54: A general test method of Johnston's SP800-22 program, which is parameterised the `examples` by the pytest configuration (listing 47).

Subsequently, we can compare the results of my randomness tests to the implementations from the same Hypothesis-generated inputs as described in section 4.1.2.

A simple Hypothesis test method is for the *Monobits* test, which has no keyword arguments and so should be determined by just the inputted sequence. This means a test method comparing different implementations only needs to be passed the sequences generated from the `mixedbits()` strategy shown in listing 48, and then assert the results of the implementations are similar enough to my randomness tests.

```
from .implementations import sgr
from .implementations import dj
...
@given(mixedbits())
def test_monobits(bits):
    result = randtests.monobits(pd.Series(bits))

    dj_result = dj.monobits(bits)
    assert isclose(result.p, dj_result.p)

    sgr_p = sgr.monobits(bits)
    assert isclose(result.p, sgr_p)
```

Listing 55: Simple comparison of Johnston's and Reid's implementation of the *Monobits* test to my own, using generated sequences from the `mixedbits()` strategy.

Other randomness tests have keyword arguments, so to compare implementations beyond the *Monobits* test, these arguments have to be generated too. Special consideration needs to be taken in bounding the length of the inputted sequence to keyword arguments relating to its size, such as `blocksize` not exceeding the sequence length `n`.

Hypothesis allows such bounded strategies to be devised through its `@composite` decorator, which exposes a `draw()` method that can extract output of a singular strategy and use it to determine subsequent strategies.

A common pattern was to draw the generated `bits` from `mixedbits()`, determine it's length `n`, and determine a `blocksize` strategy to only generate integers between `1` and `n` i.e. never exceeding the length of the sequence.

```
@st.composite
def blocks_strategy(draw, min_size=2):
    bits = draw(mixedbits(min_size=min_size))
    n = len(bits)
    blocksize = draw(st.integers(min_value=1, max_value=n))

    return bits, blocksize
...
@given(blocks_strategy(min_size=100))
def test_sgr_frequency_within_block(args):
    bits, blocksize = args

    result = randtests.frequency_within_block(bits, blocksize=blocksize)

    sgr_p = sgr.frequency_within_block(bits, blocksize=blocksize)

    assert isclose(result.p, sgr_p)
```

Listing 56: Comparing Reid's implementation of the *Frequency within Block* test to my own, using a composite strategy that returns both a sequence `bits` and a `blocksize` keyword argument. The `blocksize` strategy is bounded to the length of `bits` generated via the `draw()` method.

Comparing results of different implementations gives further confidence on the accuracy and robustness of my randomness tests. The inbuilt Hypothesis strategies are built for smaller data types however, so future modification could allow for insights into the larger sequences.

The specification of the adapted test suites could also lead to complete automation of the comparison testing. Instead of writing test methods for each randomness test and implementation(s) to compare to, all the randomness tests could be iterated over along with their respective composite strategy, and the `Implementation` and `ImplementationError` could wholly inform whether to skip a certain test suite given a specific generated test case (like in listing 54).

In the future, all open source test suites available should be adapted, but particularly those with more use (such as NIST's sts). Most are made in C, which could have their randomness tests called in Python by writing bindings. A possible metric for accuracy for a specific randomness test in coinflip could be if it gets roughly the same p-values as the majority of the other test suites.

4.2 Command-line interface

Click offers a mockable command-line interface via its `CliRunner`, which conveniently runs Click methods that represent commands. A simple smoke test is to execute commands and assert that their exit code is `0` i.e. the command executed successfully without any errors. The `coinflip example-run` command is especially useful, in that it runs all the randomness tests and so covers more functionality.

```
from click.testing import CliRunner
from coinflip import cli
...
def test_main():
    runner = CliRunner()
    result = runner.invoke(cli.main, [])

    assert result.exit_code == 0

def test_example_run():
    runner = CliRunner()
    result = runner.invoke(cli.example_run, [])

    assert result.exit_code == 0
```

Listing 57: Simple smoke tests for the `coinflip` and `coinflip example-run` commands.

To test the actual workflow of the CLI, model-based testing was employed. Hypothesis has mechanisms that can allow specification of a rules-based state machine via a class subclassing the `RuleBasedStateMachine`. The code for the final machine `CliRoutes` can be found in appendix D.

A rules-based state machine contains *bundles*, which can model certain states of the machine. A bundle called `stores` is used to model the stores (section 3.1) that should exist in the user data directory—or more precisely be recognised by the CLI—by holding the respective store names in a list.

```
from hypothesis.stateful import RuleBasedStateMachine
from hypothesis.stateful import Bundle
...
class CliRoutes(RuleBasedStateMachine):
    stores = Bundle("stores")

    def __init__(self):
        super(CliRoutes, self).__init__()

        self.runner = CliRunner()
    ...
```

Listing 58: The `CliRoutes` constructor, and `stores` bundle as a class variable. Upon initialisation a mock CLI is instantiated via Click's `CliRunner`.

The *rules* in a rules-based state machine refer to reading and writing bundles. In Hypothesis, a decorator `@rule` specifies the interaction with the bundles. Hypothesis will execute upon these rules in randomised sequences, so as to model the unpredictable nature of the CLI commands a user will use.

A rule method for initialising a store with random data via the `coinflip load` command was made. Binary sequences can be generated by the `mixedbits()` strategy (listing 48), and written to temporary files, which are then passed as the `STORE` argument to `coinflip load`. Once the store is initialised the CLI should output the store's name, which can be pushed to the `stores` bundle.

One rule is defined to see if an initialised store shows up in the `coinflip ls` command—the CLI should output the store's name, which is asserted for. Another rule removes a store via the `coinflip rm` command, and asserts that it *does not* output in the `coinflip ls` command.

Even though the rules-based state machine only contained three rules, it helped identify a great many bugs that relate to the particular *consecutive sequence* of command execution, as well as the *speed* at which they execute.

```
from tempfile import NamedTemporaryFile
from hypothesis.stateful import rule
from .randtests.strategies import mixedbits
...
r_storename = re.compile(r"Store name to be encoded "
                         r"as ([a-z\_0-9]+)\n")
...
class CliRoutes(RuleBasedStateMachine):
    ...
    @rule(target=stores, sequence=mixedbits())
    def add_store(self, sequence):
        datafile = NamedTemporaryFile()
        with datafile as f:
            for x in sequence:
                f.write(f"{x}\n")

        result = self.runner.invoke(cli.load, [datafile])
        assert result.exit_code == 0

        store_msg = r_storename.search(result.stdout)
        store = store_msg.group(1)

        return store
    ...
```

Listing 59: Simplified `add_store()` rule method in `CliRoutes`. RNG output files are mocked by use of the `mixedbits()` strategy to generate binary sequences, written to a temporary file which is loaded into an initialised store via the `coinflip load` command. The generated `store_name` is found by search the command's output `stdout` with a regular expression, saved into the `stores` bundle by being returned by the method.

```python
@rule(store=stores)
def find_store_listed(self, store):
    result = self.runner.invoke(cli.ls)
    assert re.search(store, result.stdout)
```

Listing 60: The `find_store_list()` rule method in `CliRoutes`. Using the `stores` bundle, supposedly initialised stores have their name searched in the output of the `coinflip ls` command, to assert the `coinflip` CLI recognises the store.

```python
from hypothesis.stateful import consumes
...
@rule(store=consumes(stores))
def remove_store(self, store):
    rm_result = self.runner.invoke(cli.rm, [store])
    assert rm_result.exit_code == 0

    ls_result = self.runner.invoke(cli.ls)
    assert not re.search(store, ls_result.stdout)
```

Listing 61: The `remove_store()` rule method in `CliRoutes`. Using the `stores` bundle, existing stores are removed via the `coinflip rm` command. The removed store's name is then searched in the output of the `coinflip ls` command, to assert the `coinflip` CLI *does not* recognise the store.

For example, Hypothesis generated a sequence in which `add_store()` was executed multiple times in succession and fail, because if the store names were initialised in the same second—and so have the same timestamped name—the latter would fail to initialise as a store of the same name (as it already existed). This condition was handled in the code that handles name generation `init_store()` (listing 30), and then `CliRoutes` was run again so as to *not* fail with the same rules sequence, to check the changes worked.

Due to time restrictions, rules to cover the other commands—and all their option combinations—were not implemented. The fact that the general use of Hypothesis uncovered many bugs does however suggest that doing so would be extremely useful.

```
for _ in range(3):
    timestamp = datetime.now()
    iso8601 = timestamp.strftime("%Y%m%dT%H%M%SZ")
    store_name = f"store_{iso8601}"

    if store_name not in list_stores():
        break
    else:
        sleep(1.5)
```

Listing 62: The current code in `init_store()` that handles name generation.

5 Evaluation

coinflip currently implements 9 randomness tests from SP800-22, and 5 more tests still need to be implemented. The tests are relatively simple to write, but a lot of effort was spent making the tests user-friendly, with features not present in other suites such as warnings (2.2.3) and descriptive results (2.2.1).

I personally enjoy using the command-line interface, but the unique approach of stores (3.1) needs to be tested with real end-users to see if the workflow is easier to grasp in comparison to the other suites.

The use of Hypothesis gives me confidence in the practical accuracy of coinflip's randomness tests (4.1.2). The potential of wholly comparing coinflip's randomness tests to popular suites can establish coinflip as battle-tested, even if I myself have no real background in statistics or cryptography.

So coinflip does require more work to meet my aspirations, however the groundwork laid out in this report should make further work fairly straightforward. There are many opportunities presented in randomness testing which coinflip has sought to explore, and I believe in time this project's innovative features will be invaluable for various use cases: hobbyists completely new to randomness testing; professionals testing RNGs used in cryptographic applications; and companies that desire randomness testing in their machine learning pipelines.

References

[1] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," tech. rep., Booz-allen and hamilton inc mclean va, 2001.

[2] P. L'Ecuyer and R. Simard, "Testu01: Ac library for empirical testing of random number generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, 2007.

[3] J. Reback, W. McKinney, jbrockmendel, J. V. den Bossche, T. Augspurger, P. Cloud, gfyoung, Sinhrks, A. Klein, M. Roeschke, S. Hawkins, J. Tratner, C. She, W. Ayd, T. Petersen, M. Garcia, J. Schendel, A. Hayden, MomIsBestFriend, V. Jancauskas, P. Battiston, S. Seabold, chris b1, h vetinari, S. Hoyer, W. Overmeire, alimcmaster1, K. Dong, C. Whelan, and M. Mehyar, "pandas-dev/pandas: Pandas 1.0.3," Mar. 2020.

[4] "Click." https://click.palletsprojects.com/.

[5] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[6] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and et al., "Jupyter notebooks - a publishing format for reproducible computational workflows," in *ELPUB*, 2016.

[7] P. Revesz, *Random Walk In Random And Non-random Environments*. World Scientific, 2013.

[8] user940, "Probability for the length of the longest run in n bernoulli trials." Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/59749 (version: 2014-03-11).

[9] "appdirs." https://github.com/ActiveState/appdirs.

[10] ISO, *ISO 8601:1988. Data elements and interchange formats — Information interchange — Representation of dates and times*. 1988. See also 1-page correction, ISO 8601:1988/Cor 1:1991.

[11] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, "pytest," 2004.

[12] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, p. 1891, 11 2019.

[13] "Tox." https://tox.readthedocs.io/.

[14] "Travis ci." https://travis-ci.com/.

[15] "Appveyor." https://www.appveyor.com/.

[16] D. Johnston, "sp800 22 tests." https://github.com/dj-on-github/sp800_22_tests.

[17] S. Gordon Reid, "r4nd0m." https://github.com/StuartGordonReid/r4nd0m.

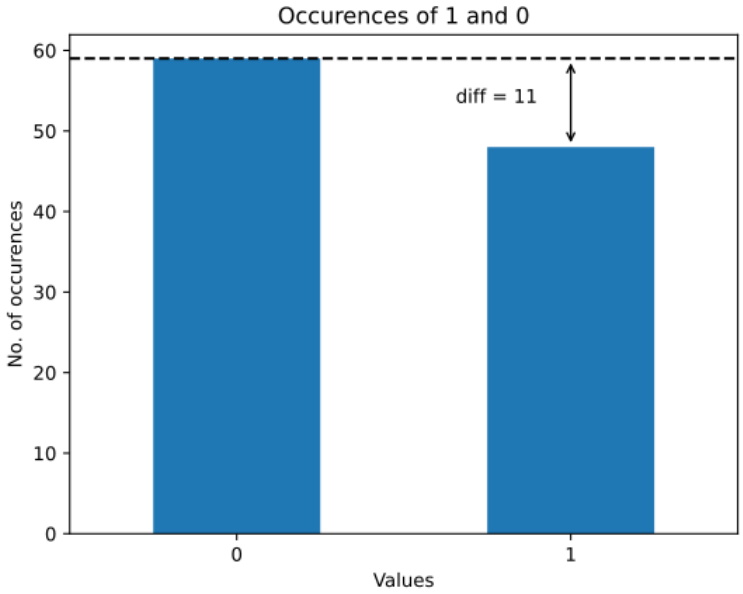
[18] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

A *Monobits* report

RNG output sample used as the *Monobits* test input:

```
10000111 10001001 00100111 00010011
11110111 00101011 00000100 10010010
10100111 11100000 11101010 01001111
10001111 111
```

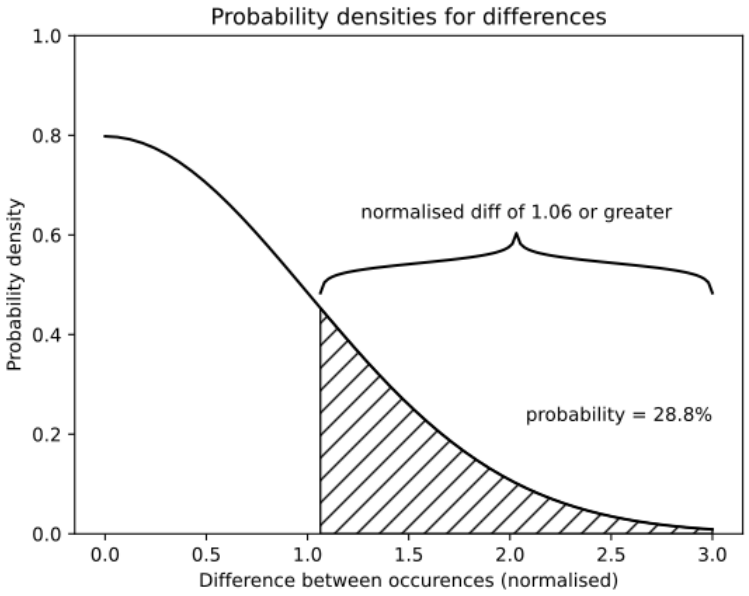
The number of occurrences for the 0 and 1 values are found and the difference is calculated.



We can compare this to the hypothetical output of a truly random RNG. A question is asked—how likely would such a RNG produce a sequence with at least a difference of 11 between the occurrences of binary values?

The likelihood would decrease with higher differences, assuming that random outputs tends towards uniformity. Such a distribution would follow a half-normal distribution (i.e. a bell-curve shape, but with it's left side flipped and added to the right).

To compare the difference of 11 with this reference distribution, we first normalise it by dividing it by the square root of the sequences length, 107. This results in a reference statistic of 1.06.



Finding the cumulative likelihood a true RNG would have such a difference or greater comes to a p-value of 0.288. The lower the p-value, the less confident we can say that this data is random.

B *Longest Run within Block* probabilities exploration

In [1]:
```
from dataclasses import dataclass
from typing import Any


@dataclass
class Run:
    value: Any
    length: int = 1

def asruns(seq):
    firstval = seq[0]
    pointer = Run(firstval, length=0)
    for value in seq:
        if value == pointer.value:
            pointer.length += 1
        else:
            yield pointer
            pointer = Run(value)
    else:
        yield pointer
```

In [2]:
```
size = 8
candidate = 1
```

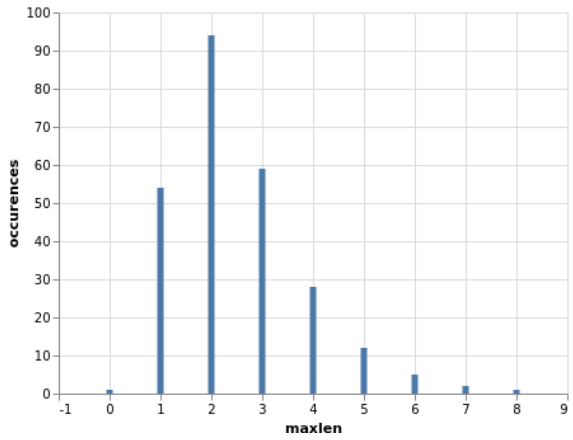
In [3]:
```
from itertools import product

maxlengths = []
for seq in product([0, 1], repeat=size):
    runs = asruns(seq)

    candidateruns = (run for run in asruns(seq) if run.value == candida
te)

    maxlen = 0
    for run in candidateruns:
        if run.length > maxlen:
            maxlen = run.length

    maxlengths.append(maxlen)
```

In [4]:
```
from collections import Counter

counts = Counter(maxlengths)
nsequences = sum(counts.values())

for maxlen, count in counts.items():
    prop = count / nsequences
    propf = round(prop, 2)
    print(f"maxlen={maxlen}, count={count}, prop={propf}")
```

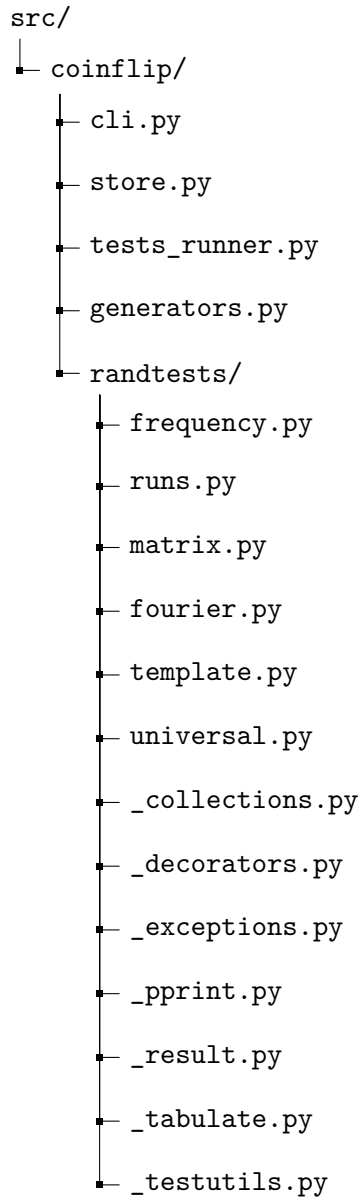
```
maxlen=0, count=1, prop=0.0
maxlen=1, count=54, prop=0.21
maxlen=2, count=94, prop=0.37
maxlen=3, count=59, prop=0.23
maxlen=4, count=28, prop=0.11
maxlen=5, count=12, prop=0.05
maxlen=6, count=5, prop=0.02
maxlen=7, count=2, prop=0.01
maxlen=8, count=1, prop=0.0
```

In [7]:
```
import pandas as pd
import altair as alt

sortedcounts = sorted(counts.items(), key=lambda item: item[0])

data = pd.DataFrame(sortedcounts, columns=["maxlen", "occurences"])

alt.Chart(data).mark_bar().encode(
    x="maxlen",
    y="occurences"
)
```

Out[7]:



C coinflip package files tree



D Command-line interface state machine

```
class CliRoutes(RuleBasedStateMachine):
    """State machine for routes taken via the CLI

    Specifies a state machine representation of the CLI to be used in
    model-based testing.

    Notes
    ---
    Read the `hypothesis stateful guide
    <https://hypothesis.readthedocs.io/en/latest/stateful.html>`_ for help on
    understanding and modifying this state machine.
    """

    def __init__(self):
        super(CliRoutes, self).__init__()

        self.runner = CliRunner()

    stores = Bundle("stores")

    @rule(target=stores, sequence=mixedbits())
    def add_store(self, sequence):
        """Mock data files and load them into initialised stores"""
        datafile = NamedTemporaryFile()
        with datafile as f:
            for x in sequence:
                x_bin = str(x).encode("utf-8")
                line = x_bin + b"\n"
                f.write(line)

            f.seek(0)
            result = self.runner.invoke(cli.load, [f.name])

        store_msg = r_storename.search(result.stdout)
        store = store_msg.group(1)

        return store
```

```
    @rule(store=stores)
    def find_store_listed(self, store):
        """Check if initialised stores are listed"""
        result = self.runner.invoke(cli.ls)
        assert re.search(store, result.stdout)

    @rule(store=consumes(stores))
    def remove_store(self, store):
        """Remove stores and check they're not listed"""
        rm_result = self.runner.invoke(cli.rm, [store])
        assert rm_result.exit_code == 0

        ls_result = self.runner.invoke(cli.ls)
        assert not re.search(store, ls_result.stdout)
```