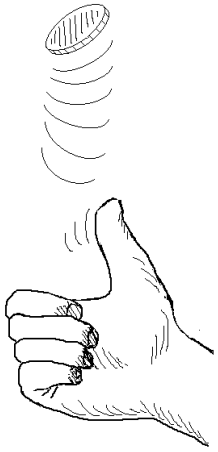
Testing a Randomness Testing Suite

Matthew Barber

August 9, 2020



Abstract

coinflip is a Python library for randomness testing, which also includes a command-line interface. This paper discusses how coinflip itself is tested.

After the introduction, the following sections detail the strategies used in testing coinflip's own randomness tests and its CLI. A final evaluation is found at the end, with a bibliography and appendix items following suit.

Note the term *SP800-22* is the publication code of the paper by NIST which specifies the randomness tests that coinflip implements [1]. The code is used as a shorthand due to it being deeply referenced.

Word count: 2458 (body text + captions)

Contents

1 Introduction

coinflip uses the *pytest* [2] package to test coinflip's own randomness tests, as well as its CLI. It also incorporates the *Hypothesis* [3] library, which provides various tools for property-based and model-based testing.

This report deviates from the recommended structure of per-quadrant sections as per the agile quadrants, due to the various unconventional testing strategies employed which acted in multiple quadrant domains to uncover bugs/problems in those multiple quadrants. The quadrants were only used to assess iterations of coinflip's overall testing strategy—not as a top-down tool to devise a plan beforehand.

Table 1: The agile quadrants covered by the sections in this report.

	Business facing		
Support team	**Q2** 2.1 Known test cases 3.2 Stateful CLI 2.3 Suite comparisons	**Q3** 2.3 Suite comparisons 3.2 Stateful CLI 3.1 Successful exits 2.2 Generating tests 3.2 Stateful CLI 4 Tooling	Critique product
	Q1	**Q4**	
	Technology facing		

2 Testing the randomness tests

Randomness tests 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.

```
>>> from coinflip.randtests import monobits
>>> help(monobits)
monobits(sequence):
    Proportion of values is compared to expected 1:1 ratio

    The difference between the frequency of the two values is found, and
    referenced to a hypothetically truly random RNG.

    Parameters
    ----------
    sequence : array-like
        Output of the RNG being tested

    Returns
    -------
    MonobitsTestResult
        Dataclass that contains the test's statistic and p-value as well as
        other relevant information gathered.
>>> from random import getrandbits
>>> sample_RNG_output = [getrandbits(1) for _ in range(5)]
>>> print(sample_RNG_output)
[0, 1, 0, 0, 1]
>>> result = monobits(sample_RNG_output)
>>> print(result)
normalised diff  0.447
p-value          0.655

value      count
-------  -------
0              3
1              2
```

Listing 1: Example usage of the *Monobits* randomness test in coinflip.

The NIST paper *Statistical test suite for random and pseudorandom number generators for cryptographic applications* [1] describes the randomness tests that coinflip implements. These tests take sampled RNG output, which would be a binary sequence. The paper will be referred to as it's internal publication code *SP800-22*.

2.1 Known test cases

Covers: ***white-box, story***

SP800-22 includes worked examples for their recommend randomness tests, which was used to help verify the implemented tests. The results of these examples are asserted to the results of coinflip's own implementations under the same parameters.

```
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 2: 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.

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 1: Walkthrough the *Monobits* test with a concrete example, provided by SP800-22.

Additionally, the working out provided in SP800-22 helped in debugging the tests, as the walkthroughs 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 test methods such as in listing 2 grew, the code became confusing to navigate which frustrated debugging. As all the examples followed a similar pattern, a container `Example` was made to specify the parameters of tests programmatically.

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

Listing 3: The named tuple `Example`, which allows specification of SP800-22 examples.

```
examples = [
    Example(
        randtest="monobits",

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

        statistic=.632455532,
        p=0.527089,
    ),
    ...
]
```

Listing 4: List of SP800-22 examples to be paramtrised by pytest, specified as `Example` tuples.

We could specify fixtures for Gherkin text files, but the use of keywords to specify the *given* (`randtest`), *when* (`bits`, `kwargs`), and *then* (`statistic`, `p`) of examples makes it easy enough to onboard non-technical users to write new test cases in the future.

2.2 Generating test cases

Covers: ***property-based, exploratory, performance, reliability***

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 library provides for.

Below is an explanation of how property-based testing works in Hypothesis, taken from their documentation [3].

> Think of a normal unit test as being something like the following:
>
> 1. Set up some data.
> 2. Perform some operations on the data.
> 3. Assert something about the result.
>
> Hypothesis lets you write tests which instead look like this:
>
> 1. For all data matching some specification.
> 2. Perform some operations on the data.
> 3. Assert something about the result.

The 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, i.e. generates data matching some specification.

A custom strategy `mixedbits()` was created to generate binary sequences in a seemingly randomised fashion. A small sample of generated sequences is available in appendix A.

Hypothesis will give "health check" errors when individual test cases take too long, i.e. if it's been more than 200 milliseconds for a test to end. It also purposely repeats sequences which were successful, to uncover any bugs which relate to global states.

```
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 5: 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.

Just running supposedly valid data on the implemented randomness tests and testing there are no warnings is very useful. For example, a bug was discovered in the *Discrete Fourier Transform* test by using 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 discovered thanks to the generated boundary case, 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 6: The `NonBinarySequenceError` and derivative `NonBinaryTruncatedSequenceError` being raised when using the *Discrete Fourier Transform* randomness test.

2.3 Comparing to other suites

Covers: ***boundary, equivalence, comparison, exploratory***

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 [4], and the other by Stuart Gordon Reid [5].

Adapting these suites to mirror coinflip's own API enabled running the SP800-22 examples on them (section 2.1). If SP800-22's examples were taken as completely accurate, then it could lead to false-positives if it does infact have mistakes. When coinflip fails to match an example's conclusion, the other suites can be checked first to see if they reach they fail as well.

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 helped identify a problem in the same SciPy [6] fast Fourier transform method we were all using.

Table 2: 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

Subsequently, we can compare the results of coinflip's randomness tests to the implementations from the same Hypothesis-generated inputs (section 2.2).

Hypothesis generates sequences with numerous rubrics, including purposely generating boundary cases, and purposely generating cases that fit into automatically identified equivalence partitions (e.g. classes "sequence length" and "0-to-1 ratios"). By comparing the results of other suites, we can verify all these cases are actually handled correctly.

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 5, 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 7: Simple comparison of Johnston's and Reid's implementation of the *Monobits* test to my own, using generated sequences from the `mixedbits()` strategy.

Comparing results of different implementations gives further confidence on the accuracy and robustness of my randomness tests. In the future all open source test suites available should be adapted; 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.

3 Testing the command-line interface

coinflip's CLI consists of subcommands to the `coinflip` group command. The subcommands provide various features to aid users in loading their RNG output samples and subsequently running the randomness tests on respective samples.

```
$ coinflip --help
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.

...

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

Commands:
example-run  Run randomness tests on example data.
load         Loads DATA into a store.
run          Run randomness tests on data in STORE.
ls           List all stores.
rm           Delete STORE.
...
```

Listing 8: Simplified description of the `coinflip` command.

3.1 Successful exit codes

Covers: ***smoke***

coinflip's command-line interface uses the Click [7] package, which offers a mockable command-line interface via its `CliRunner` to conveniently run CLI 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.

```
$ coinflip example-run --help
Usage: coinflip example-run [OPTIONS]

Run randomness tests on example data.
```

Listing 9: Simplified description of the `coinflip example-run` command.

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

3.2 Stateful testing

Covers: ***model-based, functional, exploratory, reliability***

coinflip's CLI allows users to load their data into *stores*, where each store pertains to a specific RNG output sample. In actuality, the stores are simply folders which are located in the user's OS-specific user data directory (e.g. `%AppData%` in Windows); coinflip is providing an abstraction to these folders.

One can load their RNG output sample via the `coinflip load` command; can view the stores currently in their user data directory via the `coinflip ls` command; and can remove stores via the `command rm` command.

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

Listing 11: 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.

To test these abstraction are robust, model-based testing was employed. Hypothesis has mechanisms that can allow specification of a rules-based state machine (by inheritance of its `RuleBasedStateMachine` class), which are "run" in seemingly random ways. A small sample of generated state machine runs is available in appendix B.

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 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.

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

Listing 12: 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 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 13: 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 5), 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.

```
from tempfile import NamedTemporaryFile
from hypothesis.stateful import rule
from .randtests.strategies import mixedbits
...
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)  # "Store name to be encoded as <1>"

        return store
    ...
```

Listing 14: 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` is found by search the command's output `stdout` with a regular expression, saved into the `stores` bundle by being returned by the method.

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.

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

Listing 15: 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.

```
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 16: 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.

In the future, rules to cover the other commands—and all their option combinations—should be implemented. The fact that the general use of Hypothesis uncovered many bugs does suggest that doing so would be extremely useful.

4 Tooling

Covers: ***regression, compatibility, coverage***

The Python testing wrapper tox [8] allows the specification of various testing profiles, including the multiple Python versions one wishes to execute them in. This meant a short command `tox -e py37` could be specified to ensure changes did not break the expected behaviour of coinflip.

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

Listing 17: Command that `tox -e py37` wraps.

coinflip uses continuous integration which automatically runs these tox profiles, employing the Travis CI [9] and AppVeyor [10] services. tox allows specifying all the tests described in this report to be executed together

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 in was uncovering a problem with the creation of the user data directory, which happens when loading in stores for the first time. I used the correct OS-specific user data directories for both Linux and Windows, but did not specify the "app author" which only Windows uses to construct these directories—because I ran the tests on my Linux machine only, it was only by using cross-platform CI that I could discover and remedy this problem.

coinflip also utilises pytest's coverage capabilities, which provides insight into what code is actually being run when using tests. This enables insight into what tests should be developed in the future, so as to be more confident that future changes do not break existing functionality.

80.2% of code is covered right now, with missing tests mainly residing in the CLI-related code files. This would be remedied by expanding the functionality of the CLI state machine (section 3.2), and progress of which can be visually analysed by starburst charts over time.

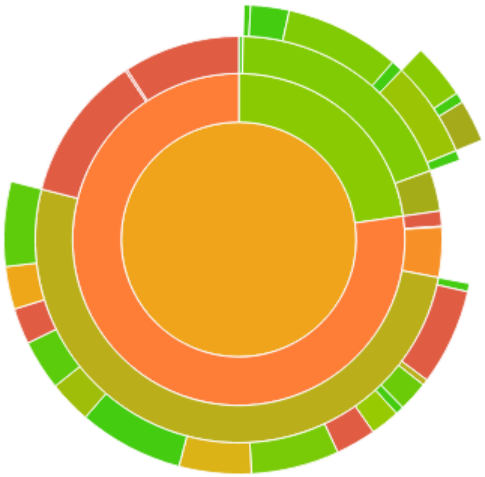


Figure 2: Starburst representation of coinflip's current test coverage, primarily identify a lack of coverage for CLI functionality.

5 Evaluation

I believe the test strategy of coinflip covers is a good foundation, particularly in the potential of wholly comparing coinflip's randomness tests to other suites can establish coinflip as battle-tested—even if I myself have no real background in statistics or cryptography.

The Q3 aspect is only fulfilled by the exploratory nature of the Hypothesis library. A study with real users would be useful in the assessing coinflip's user experience by observing actual human behaviour.

Code repository: https://github.com/Honno/coinflip/
Documentation: https://coinflip.readthedocs.io/
Video demo: https://www.youtube.com/watch?v=0xrWG3Ki9Z8
(testing strategy begins at 11:39)

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

[3] 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.

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

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

[6] 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.

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

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

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

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

A Sample of generated binary sequences

Sample of the consecutive binary sequences (represented per-line) run on coinflip's randomness tests.

```
01
10
110
111100110001
010
01000
110
11001110
010
0101
100
0011011000001
0011011000001
0011011100001
0010011100001
0010011100001
0010011100001
0011011100001
1110101111
0011001
0011001
1011001
1011101
101
100
10100
00110101010
0110101010
0110101010
0110101010
0110101010
0110101010
011
1111001
1001
01
011110
011111
011111
000110
0001100
0001101
0001101
0001101
0011101
0001101
10010001111110100110
10010001111110110110
10010001111110110110
10010001011110110110
10010001011110110110
```

```
10010001011110110110
10010001011110111110
11001
1100
0100
001100
001100
000100
11101110101
011
111010010100
111010010101
111010010101
111010000101
11101000010110
1110100001011000110101110010
11000100111000
11000100111000
11000100011000
11000100011000
00001
01001
010
01011
0101110000
01011100000100
0101010000100
000011111110000
000011111110010
000011111110010
000011111111010
000010111111010
000010111111010
000010111111010
11110001111
11110101111
11110101111
11110101111
11110101111
11110100111
01110100111
011011001010011111
0110110010100111110100100100011010
0110110010100111111100100100011010
0110110010110111111100100100011010
0110110010111111111100100100011010
0110110010111111111100100100011010
0110110010111111111100100100011110
101
...and so on
```

B Sample of generated CLI state machine runs

```
>>> SETUP new state machine
    LOAD store1 with sequence 10
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    LOAD store1 with sequence 10010
    LOAD store2 with sequence 11011
    REMOVE store1
    LS stores, check store1 not listed
    TEARDOWN state machine
>>> SETUP new state machine
    LOAD store1 with sequence 11111001010
    LOAD store2 with sequence 00101
    REMOVE store1
    LS stores, check store1 not listed
    REMOVE store2
    LS stores, check store2 not listed
    LOAD store3 with sequence 001111
    LS stores, check store3 listed
    LOAD store4 with sequence 011011000011001
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    LOAD store1 with sequence 1000011001001000010
    LOAD store2 with sequence 001100010100
    LOAD store3 with sequence 111110111001
    LOAD store4 with sequence 1011101000110111
    LOAD store5 with sequence 1010
    LOAD store6 with sequence 0101101
    LOAD store7 with sequence 11111010001100
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    LOAD store1 with sequence 11100
    LOAD store2 with sequence 10010
    LOAD store3 with sequence 1000011110
    LOAD store4 with sequence 01111
    REMOVE store1
    LS stores, check store1 not listed
    REMOVE store4
    LS stores, check store4 not listed
    LOAD store5 with sequence 011010101000000111
    TEARDOWN state machine
```

```
>>> SETUP new state machine
    LOAD store1 with sequence 011100100
    REMOVE store1
    LS stores, check store1 not listed
    LOAD store2 with sequence 1011100001
    LOAD store3 with sequence 000000111110
    REMOVE store2
    LS stores, check store2 not listed
    LOAD store4 with sequence 00000000100011011100101100101101101
    REMOVE store4
    LS stores, check store4 not listed
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    LOAD store1 with sequence 11000111101101111011000
    LOAD store2 with sequence 11101011010101
    LS stores, check store1 listed
    LOAD store3 with sequence 1101
    LOAD store4 with sequence 011110110001011
    REMOVE store2
    LS stores, check store2 not listed
    LOAD store5 with sequence 1101001001010100
    LOAD store6 with sequence 11001011
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    TEARDOWN state machine
>>> SETUP new state machine
    LOAD store1 with sequence 00101
    REMOVE store1
    LS stores, check store1 not listed
    LOAD store2 with sequence 10111000101001101100
    LOAD store3 with sequence 10011010000000100001011010
    REMOVE store3
    LS stores, check store3 not listed
    LOAD store4 with sequence 011
    LOAD store5 with sequence 11110100
    TEARDOWN state machine
    ...and so on
```