Falsify your Software: validating scientific code with property-based testing

Zac Hatfield-Dodds‡*

Abstract—Where traditional example-based tests check software using manually-specified input-output pairs, property-based tests exploit a general description of valid inputs and program behaviour to automatically search for falsifying examples. Given that Python has excellent property-based testing tools, such tests are often *easier* to work with and routinely find serious bugs that all other techniques have missed.

I present four categories of properties relevant to most scientific projects, demonstrate how each found real bugs in Numpy and Astropy, and propose that property-based testing should be adopted more widely across the SciPy ecosystem.

Index Terms—methods, software, validation, property-based testing

Introduction

Much research now depends on software for data collection, analysis, and reporting; including on software produced and maintained by researchers. This has empowered us enormously: it is hard to imagine an analysis that was possible *at all* a generation ago which could not be accomplished quickly by a graduate student today.

Unfortunately, this revolution in the power and sophistication of our software has largely outstripped work on validation. While it would be unthinkable to publish research based on custom-built and unvalidated physical instruments, this is routine in software. As an effect, [Soe15] estimates that

> Any reported scientific result could very well be wrong if data have passed through a computer, and these errors may remain largely undetected[1]. It is therefore necessary to greatly expand our efforts to validate scientific software.

I argue that property-based testing [Mac] is more effective for validation of Python programs than using only traditional example-based tests[2], and support this argument with a variety of examples from well-known scientific Python projects.

This is a recent development: while the concept of property-based testing dates back to 1999 [CH00], early tools required deep computer-science expertise. Since 2015, Hypothesis [MHDC19] has made state-of-the-art testing technology available *and accessible to non-experts* in Python, and has added multiple features designed for testing scientific programs[3] since 2019.

What is property-based testing?

Where example-based tests check for an exact expected output, property-based tests make *less precise* but *more general* assertions. By giving up hand-specification of the expected output, we gain tests that can be run on a wide range of inputs.

This generality also guides our tests to the right level of abstraction, and gives clear design feedback: where example-based tests map one input to one output regardless of complexity, every special case or interacting feature has to be addressed. Clean abstractions which allow you to say "for all ...", without caveats, stand in clear contrast and are a pleasure to test.

Tests which use random data are usually property-based, but using a library designed for the task has several advantages:

- a concise and expressive interface for describing inputs
- tests are never flaky - failing examples are cached and replayed, even if the test failed on a remote build server
- automatic shrinking facilitates debugging by presenting a minimal failing example for each distinct error

Automating these tedious tasks makes finding bugs considerably faster, and debugging both easier and more fun. When Hypothesis world-class shrinker [MD] hands you a failing example, you *know* that every feature is relevant - if any integer could be smaller (without the test passing) it would be; if any list could be shorter or sorted (ditto) it would be, and so on.

Why is this so effective?

Examples I wouldn't think of reveal bugs I didn't know were possible. It turns out that general descriptions of your data have several advantages over writing out specific examples, and that these are even stronger for research code.

Input descriptions are concise. Writing out a Hypothesis "strategy"[4] describing objects like a Numpy array or a Pandas dataframe is often less code than a single instance, and clearly expresses to readers what actually matters. The sheer tedium of writing out representative test cases, without resorting to literally random data, is a substantial deterrent to testing data analysis code; with property-based testing you don't have to.

* *Corresponding author: zac.hatfield.dodds@anu.edu.au*
‡ *Australian National University*

Copyright © 2020 Zac Hatfield-Dodds. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1. and indeed [BNNL+19] reported such a bug, affecting around 160 papers.
2. i.e. common workflows using `pytest`. If you have no automated tests at all, fix that first, but your second test could reasonably use Hypothesis.
3. e.g. numeric-aware error reporting, first-class support for array shapes including broadcasting and gufunc signatures, dtypes and indexers, etc.

The system is designed to find bugs. Hypothesis also comes packed with well-tuned heuristics and tools for finding bugs which uniformly random data would almost never find - literal 'edge cases' in the space of possible inputs. In one memorable case a new user was testing coordinate transformation logic, which fails for singular matrices (a set of measure zero). Hypothesis knows nothing at all about matrices, or even the topic of the test, but promptly generated a failing example anyway.

Describe once, test everywhere. In a codebase with M variations on the core data structures and N features, example-based tests have $M \times N$ tests to write - and it's all too easy to forget to test the interaction of lesser-known features[5]. With property-based tests those M variations can be designed into the strategies which describe your data, scaling as $M + N$ and ensuring that nothing is forgotten[6].

This scaling effect makes effective testing much easier for new contributors, who do not need to consider all possible feature interactions - they will arise naturally from the shared input descriptions. For example, Hypothesis' Numpy extension includes tools to describe arrays, array shapes including broadcasting and generalised ufunc signatures, scalar and structured dtypes, and both basic and advanced indexing. Thinking carefully about what inputs *should* be supported is usually a valuable exercise in itself!

We face multiple sources of uncertainty. When experimental results come out weird, unpicking the unexpected behaviour of your research domain from the possibility of equipment error or software bugs is hard enough already. Property-based tests let you verify more general behaviours of your code, and focus on the domain rather than implementation details.

Properties and Case Studies

In this section I present four categories of properties. While not an exhaustive list[7], they are relevant to a very wide range of scientific software - and when tested often uncover serious errors.

I also present case studies of real-world bugs[8] from the SciPy stack, especially from foundational libraries like Numpy [Oli06] and Astropy [ART+13] [PWSG+18]. While seriously under-resourced given their importance to essentially all research in their fields [Num] [ea16], they are well-engineered and no more defect-prone than any comparable software. *If it can happen to them, it can certainly happen to you.*

The bugs presented below were each discovered, reported, and fixed within a few days thanks to a community-driven and open source development model; and projects from Astropy to Xarray - via Numpy and Pandas - have begun to adopt property-based tests.

Outputs within expected bounds

For many functions, the simplest property to check is that their output is within some expected bound. These may be computational or logical bounds like the limits of probability as $[0,1]$, or might be physical bounds like the temperature $-273.15°C$.

Consider the `softmax` function, as described by the SciPy documentation[9]. This function is often used to convert a vector of real numbers into a probability distribution, so we know that sum should always be (approximately) equal to one. Let's test that with an example-based and a property-based test:

```
from hypothesis import given, strategies as st
import hypothesis.extra.numpy as npst

def softmax(x):
    return np.exp(x) / np.exp(x).sum()

def test_softmax_example():
    assert softmax(np.arange(5)).sum() == 1

@given(npst.arrays(
    dtype=float,
    shape=npst.array_shapes(),
    elements=st.floats(
        allow_nan=False, allow_infinity=False
    )
))
def test_softmax_property(arr):
    total = softmax(arr).sum()
    np.testing.assert_almost_equal(total, 1)
```

While our example-based test passes for small arrays of small integers, the naive algorithm is numerically unstable! Our property-based test fails almost instantly, showing us the minimal example of overflow with `np.exp([710.])`. If we instead use `np.exp(x - x.max())`, the test passes.

I will not argue that this kind of testing can substitute for numerical analysis, but rather that it can easily be applied to routines which would otherwise not be analysed at all.

A more sophisticated example of bounds testing comes from recent work in Astropy[10], using Hypothesis to check that conversions between different time scales did not unexpectedly lose precision[11]. Astropy contributors wrote custom strategies to incorporate bias towards leap-seconds (unrepresentable in `datetime.datetime`), and an `assert_almost_equal` helper which uses `hypothesis.target()` to guide the search process towards larger errors.

These tests found that round-trip conversions could be off by up to twenty microseconds over several centuries[12] due to loss of precision in `datetime.timedelta.total_seconds()`. This effort also contributed to improved error reporting around the 'threshold problem'[13], where a minimal failing example does not distinguish between subtle and very serious bugs.

Round-trip properties

Whenever you have a pair of inverse functions, think of round-trip testing. If you save and then load data, did you lose any? If

4. for historical reasons, Hypothesis calls input descriptions 'strategies'

5. e.g. signalling NaNs, zero-dimensional arrays, structured Numpy dtypes with field titles in addition to names, explicit dtype padding or endianness, etc. Possible *combinations* of such features are particularly neglected.

6. test 'fixture' systems scale similarly, but are less adaptable to individual tests and can only be as effective as the explicit list of inputs they are given.

7. a notable omission is the 'null property', where you execute code on valid inputs but do not make any assertions on its behaviour. This is shockingly effective at triggering internal errors, even before use of assertions in the code under test - and a simple enough technique to explain in a footnote!

8. preferring those which can be demonstrated and explained in only a few lines, though we have found plenty more which cannot.

9. docs.scipy.org/doc/scipy/reference/generated/scipy.special.softmax.html

10. culminating in github.com/astropy/astropy/pull/10373

11. as background, Python's builtin `datetime.datetime` type represents time as a tuple of integers for year, month, ..., seconds, microseconds; and assumes UTC and the current Gregorian calendar extended in both directions. By contrast `astropy.time.Time` represents time with a pair of 64-bit floats; supports a variety of civil, geocentric, and barycentric time scales; and maintains sub-nanosecond precision over the age of the universe!

12. while a 20us error might not sound like much, it is a *hundred billion times* the quoted precision, and intolerable for e.g. multi-decade pulsar studies.

13. described in hypothesis.works/articles/threshold-problem/ and addressed by github.com/HypothesisWorks/hypothesis/pull/2393

you convert between two formats, or coordinate systems, can you convert back?

These properties are remarkably easy to test, vitally important, and often catch subtle bugs due to the complex systems interactions. It is often worth investing considerable effort to describe *all* valid data, so that examples can be generated with very rare feature combinations.

If you write only one test based on this paper, *try to save and load any valid data*.

I have consistently found testing IO round-trips to be among the easiest and most rewarding property tests I write. My own earliest use of Hypothesis came after almost a month trying to track down data corruption issues in multi-gigabyte PLY files. Within a few hours I wrote a strategy to generate PLY objects, executed the test, and discovered that our problems were due to mishandling of whitespace in the file header[14].

Even simple tests are highly effective though - consider as an example

```
@given(st.text(st.characters())
       .map(lambda s: s.rstrip("\x00")))
def test_unicode_arrays_property(string):
    assert string == np.array([string])[0]
```

This is a more useful test that it might seem: after working around null-termination of strings, we can still detect a variety of issues with length-aware dtypes, Unicode version mismatches, or string encodings. A very similar test did in fact find an encoding error[15], which was traced back to a deprecated - and promptly removed - compatibility workaround to support 'narrow builds' of Python 2.

Differential testing

Running the same input through your code and through a trusted - or simply different - implementation is another widely applicable property: any difference in the outputs indicates that *at least* one of them has a bug. Common sources of alternative implementations include:

Another project or language. If you aim to duplicate functionality from an existing project, you can check that your results are identical for whatever overlap exists in the features of the two projects. This might involve cross-language comparisons, or be as simple as installing an old version of your code from before a significant re-write.

A toy or brute-force implementation which only works for small inputs might be out of the question for 'production' use, but can nonetheless be useful for testing. Alternatively, differential testing can support ambitious refactoring or performance optimisations - taking existing code with "obviously no bugs" and using it to check a faster version with "no obvious bugs".

Varying unrelated parameters such as performance hints which are not expected to affect the calculated result. Combining this and the previous tactic, try comparing single-threaded vs. multi-threaded mode - while some care is required to ensure determinism it is often worth the effort.

As our demonstration, consider the `numpy.einsum` function and two tests. The example-based test comes from the Numpy test suite; and the property-based test is a close translation - it still requires two-dimensional arrays, but allows the shapes and contents to vary. Note that both are differential tests!

```
def test_einsum_example():
    p = np.ones(shape=(10, 2))
    q = np.ones(shape=(1, 2))
    assert_array_equal(
        np.einsum("ij,ij->j", p, q, optimize=True),
        np.einsum("ij,ij->j", p, q, optimize=False)
    )

@given(
    data=st.data(),
    dtype=npst.integer_dtypes(),
    shape=npst.array_shapes(min_dims=2, max_dims=2),
)
def test_einsum_property(data, dtype, shape):
    p = data.draw(npst.arrays(dtype, shape))
    q = data.draw(npst.arrays(dtype, shape))
    assert_array_equal( ... )  # as above
```

When an optimisation to avoid dispatching to `numpy.tensordot` over a dimension of size one was added, the example-based test kept passing - despite the bug `if 1 in operands[n]` instead of `if 1 in operands[n].shape`[16]. This bug could only be triggered with `optimize=True` and an input array with a dimension of size one, *xor* containing the integer `1`. This kind of interaction is where property-based testing really shines.

There's another twist to this story though: the bug was actually identified downstream of Numpy, when Ryan Soklaski was testing that `Tensors` from his auto-differentiation library `MyGrad` [Sok] were in fact substitutable for Numpy arrays[17]. He later said of property-based tests that[18]

> It would have been impossible for me to implement a trustworthy autograd library for my students to learn from and contribute to if it weren't for Hypothesis.

Metamorphic properties

A serious challenge when testing research code is that the correct result may be genuinely unknown - and running the shiny new simulation or analysis code is the only way to get any result at all. One very powerful solution is to compare several input-output pairs, instead of attempting to analyse one in isolation:

> A test oracle determines whether a test execution reveals a fault, often by comparing the observed program output to the expected output. This is not always practical... Metamorphic testing provides an alternative, where correctness is not determined by checking an individual concrete output, but by applying a transformation to a test input and observing how the program output "morphs" into a different one as a result. [SFSR16]

Let's return to `softmax` as an example. We can state general properties about a single input-output pair such as "all elements of the output are between zero and one", or "the sum of output elements is approximately equal to one"[19]. A metamorphic property we could test is scale-invariance: multiplying the input elements by a constant factor should leave the output approximately unchanged.

```
@given(arr=..., factor=st.floats(-1000, 1000))
def test_softmax_metamorphic_property(arr, factor):
```

14. github.com/dranjan/python-plyfile/issues/9
15. github.com/numpy/numpy/issues/15363
16. github.com/numpy/numpy/issues/10930
17. making `test_einsum_property` a differential test derived from a derivative auto-differentiator.
18. github.com/HypothesisWorks/hypothesis/issues/1641

```
result = softmax(arr)
scaled = softmax(arr * factor)
np.testing.assert_almost_equal(result, scaled)
```

Astropy's tests for time precision include metamorphic as well as round-trip properties: several assert that given a `Time`, adding a tiny timedelta then converting it to another time scale is almost equal to converting and then adding.

Metamorphic properties based on domain knowledge are particularly good for testing "untestable" code. In bioinformatics, [CHLX09] presents testable properties for gene regulatory networks and short sequence mapping[20], and found a bug attributable to the *specification* - not just implementation errors. METTLE [XZC+20] proposes eleven generic metamorphic properties for unsupervised machine-learning sytems[21], and studies their use as an aid to end-users selecting an appropriate algorithm in domains from LIDAR to DNA sequencing.

Conclusion

Example-based tests provide anecdotal evidence for validity, in that the software behaves as expected on a few known and typically simple inputs. Property-based tests require a precise description of possible inputs and a more general specification, but then automate the search for falsifying counter-examples. They are quick to write, convenient to work with, and routinely find serious bugs that all other techniques had missed.

I argue that this Popperian approach is superior to the status quo of using only example-based tests, and hope that the property-based revolution comes quickly.

Acknowledgements

Thanks to David MacIver and the many others who have contributed to Hypothesis; to Hillel Wayne, Kathy Reid, and Ryan Soklaski for their comments on an early draft of this paper; to Anne Archibald for her work with threshold tests; and to the many mantainers of the wider Python ecosystem.

References

[ART+13] Astropy Collaboration, T. P. Robitaille, E. J. Tollerud, P. Greenfield, M. Droettboom, E. Bray, T. Aldcroft, M. Davis, A. Ginsburg, A. M. Price-Whelan, W. E. Kerzendorf, A. Conley, N. Crighton, K. Barbary, D. Muna, H. Ferguson, F. Grollier, M. M. Parikh, P. H. Nair, H. M. Unther, C. Deil, J. Woillez, S. Conseil, R. Kramer, J. E. H. Turner, L. Singer, R. Fox, B. A. Weaver, V. Zabalza, Z. I. Edwards, K. Azalee Bostroem, D. J. Burke, A. R. Casey, S. M. Crawford, N. Dencheva, J. Ely, T. Jenness, K. Labrie, P. L. Lim, F. Pierfederici, A. Pontzen, A. Ptak, B. Refsdal, M. Servillat, and O. Streicher. Astropy: A community Python package for astronomy. *Astronomy & Astrophysics*, 558, October 2013. doi:10.1051/0004-6361/201322068.

[BNNL+19] Jayanti Bhandari Neupane, Ram P. Neupane, Yuheng Luo, Wesley Y. Yoshida, Rui Sun, and Philip G. Williams. Characterization of leptazolines a–d, polar oxazolines from the cyanobacterium leptolyngbya sp., reveals a glitch with the "willoughby–hoye" scripts for calculating nmr chemical shifts. *Organic Letters*, 21(20):8449–8453, 2019. URL: https://doi.org/10.1021/acs.orglett.9b03216, doi:10.1021/acs.orglett.9b03216.

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000. URL: https://doi.org/10.1145/357766.351266, doi:10.1145/357766.351266.

[CHLX09] Tsong Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1):24, 2009. URL: https://doi.org/10.1186/1471-2105-10-24, doi:10.1186/1471-2105-10-24.

[ea16] Demitri Muna et al. The astropy problem, 2016. arXiv:1610.03159.

[LSH+] Benjamin Lee, Reva Shenwai, Zongyi Ha, Michael B. Hall, and Vaastav Anand. Hypothesis-Bio. URL: https://github.com/Lab41/hypothesis-bio.

[Mac] David MacIver. In Praise of Property-Based Testing. URL: https://increment.com/testing/in-praise-of-property-based-testing/.

[MD] David MacIver and Alastair Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. to be published in https://2020.ecoop.org/details/ecoop-2020-papers/13/Test-Case-Reduction-via-Test-Case-Generation-Insights-From-the-Hypothesis-Reducer. URL: https://drmaciver.github.io/papers/reduction-via-generation-preview.pdf.

[MHDC19] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019. URL: https://doi.org/10.21105/joss.01891, doi:10.21105/joss.01891.

[Num] NumFocus. Why is numpy only now getting funded? URL: https://numfocus.org/blog/why-is-numpy-only-now-getting-funded.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[PWSG+18] A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, M. W. Craig, N. Dencheva, and et al. The astropy project: Building an open-science project and status of the v2.0 core package. *The Astronomical Journal*, 156(3):123, Aug 2018. URL: http://dx.doi.org/10.3847/1538-3881/aabc4f, doi:10.3847/1538-3881/aabc4f.

[SFSR16] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[Soe15] David A. W. Soergel. Rampant software errors may undermine scientific results. *F1000Research*, 3:303, July 2015. URL: https://doi.org/10.12688/f1000research.5930.2, doi:10.12688/f1000research.5930.2.

[Sok] Ryan Soklaski. Mygrad. URL: https://github.com/rsokl/MyGrad.

[XZC+20] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. Mettle: A metamorphic testing approach to assessing and validating unsupervised machine learning systems. *IEEE Transactions on Reliability*, page 1–30, 2020. URL: http://dx.doi.org/10.1109/TR.2020.2972266, doi:10.1109/tr.2020.2972266.

19. or even `np.argsort(arr) == np.argsort(softmax(arr))`

20. e.g. K-means clustering

21. for which `hypothesis-bio` [LSH+] provides many useful data-generation strategies