<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Quick start guide</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>An example</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Installing</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Running tests</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Writing tests</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Where to start</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Community</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Code of conduct</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>The Purpose of Hypothesis</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Details and advanced features</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Making assumptions</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>Settings</td>
<td>13</td>
</tr>
<tr>
<td>4.3</td>
<td>Defining strategies</td>
<td>15</td>
</tr>
<tr>
<td>4.4</td>
<td>The gory details of given parameters</td>
<td>15</td>
</tr>
<tr>
<td>4.5</td>
<td>Custom function execution</td>
<td>16</td>
</tr>
<tr>
<td>4.6</td>
<td>Using Hypothesis to find values</td>
<td>17</td>
</tr>
<tr>
<td>4.7</td>
<td>Providing explicit examples</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>What you can generate and how</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Infinite streams</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>Mapping</td>
<td>24</td>
</tr>
<tr>
<td>5.3</td>
<td>Filtering</td>
<td>24</td>
</tr>
<tr>
<td>5.4</td>
<td>Chaining strategies together</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Additional packages</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>hypothesis-datetime</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>hypothesis-fakefactory</td>
<td>32</td>
</tr>
<tr>
<td>6.3</td>
<td>hypothesis-pytest</td>
<td>33</td>
</tr>
<tr>
<td>6.4</td>
<td>hypothesis-django</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>Stateful testing</td>
<td>35</td>
</tr>
<tr>
<td>7.1</td>
<td>Rule based state machines</td>
<td>35</td>
</tr>
<tr>
<td>7.2</td>
<td>Generic state machines</td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>Compatibility</td>
<td>41</td>
</tr>
<tr>
<td>8.1</td>
<td>Python versions</td>
<td>41</td>
</tr>
</tbody>
</table>
8.2 Operating systems ......................................................... 41
8.3 Testing frameworks ....................................................... 41
8.4 Regularly verifying this ................................................... 42

9 Some more examples ......................................................... 43
9.1 How not to sort by a partial order ..................................... 43
9.2 Time zone arithmetic ..................................................... 45
9.3 Condorcet’s Paradox ...................................................... 47
9.4 Fuzzing an HTTP API .................................................... 48

10 Innovative features of Hypothesis ..................................... 51
10.1 Templating ................................................................. 51
10.2 Multi-stage simplification .............................................. 52
10.3 Parametrization .......................................................... 53
10.4 The database ............................................................. 54
10.5 Example tracking ........................................................ 54
10.6 The strategy function .................................................... 54

11 Changelog ................................................................. 55
11.1 1.4.0 - 2015-05-04 .................................................... 55
11.2 1.3.0 - 2015-04-22 ...................................................... 56
11.3 1.2.1 - 2015-04-16 ..................................................... 57
11.4 1.2.0 - 2015-04-15 ..................................................... 57
11.5 1.1.1 - 2015-04-07 ..................................................... 57
11.6 1.1.0 - 2015-04-06 ..................................................... 58
11.7 1.0.0 - 2015-03-27 ..................................................... 58
11.8 0.9.2 - 2015-03-26 ..................................................... 58
11.9 0.9.1 - 2015-03-25 ..................................................... 58
11.10 0.9.0 - 2015-03-23 ................................................... 59
11.11 0.7.2 - 2015-03-22 ................................................... 59
11.12 0.7.1 - 2015-03-21 ................................................... 59
11.13 0.7.0 - 2015-03-20 ................................................... 59
11.14 0.6.0 - 2015-03-13 ................................................... 60
11.15 0.5.0 - 2015-02-10 ................................................... 60
11.16 0.4.3 - 2015-02-05 ................................................... 61
11.17 0.4.2 - 2015-02-04 ................................................... 61
11.18 0.4.1 - 2015-02-03 ................................................... 62
11.19 0.4.0 - 2015-01-21 ................................................... 62
11.20 0.3.2 - 2015-01-16 ................................................... 63
11.21 0.3.1 - 2015-01-13 ................................................... 63
11.22 0.3.0 - 2015-01-12 ................................................... 64
11.23 0.2.2 - 2015-01-08 ................................................... 64
11.24 0.2.1 - 2015-01-07 ................................................... 64
11.25 0.2.0 - 2015-01-07 ................................................... 64
11.26 0.1.4 - 2013-12-14 ................................................... 65
11.27 0.1.3 - 2013-05-03 ................................................... 65
11.28 0.1.2 - 2013-03-24 ................................................... 65
11.29 0.1.1 - 2013-03-24 ................................................... 65
11.30 0.1.0 - 2013-03-23 ................................................... 65
11.31 0.0.5 - 2013-03-13 ................................................... 65
11.32 0.0.4 - 2013-03-13 ................................................... 65
11.33 0.0.3 - 2013-03-13 ................................................... 66
11.34 0.0.2 - 2013-03-12 ................................................... 66
11.35 0.0.1 - 2013-03-10 ................................................... 66
Hypothesis is a Python library for creating unit tests which are simpler to write and more powerful when run, finding edge cases in your code you wouldn’t have thought to look for. It is stable, powerful and easy to add to any existing test suite.

It works by letting you write tests that assert that something should be true for every case, not just the ones you happen to think of.

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.

This is often called property based testing, and was popularised by the Haskell library Quickcheck.

It works by generating random data matching your specification and checking that your guarantee still holds in that case. If it finds an example where it doesn’t, it takes that example and cuts it down to size, simplifying it until it finds a much smaller example that still causes the problem. It then saves that example for later, so that once it has found a problem with your code it will not forget it in future.

Writing tests of this form usually consists of deciding on guarantees that your code should make - properties that should always hold true, regardless of what the world throws at you. Examples of such guarantees might be:

- Your code shouldn’t throw an exception, or should only throw a particular type of exception (this works particularly well if you have a lot of internal assertions).
- If you delete an object, it is no longer visible.
- If you serialize and then deserialize a value, then you get the same value back.

Now you know the basics of what Hypothesis does, the rest of this documentation will take you through how and why. It’s divided into a number of sections, which you can see in the sidebar (or the menu at the top if you’re on mobile), but you probably want to begin with the Quick start guide, which will give you a worked example of how to use Hypothesis and a detailed outline of the things you need to know to begin testing your code with it.
Quick start guide

This document should talk you through everything you need to get started with Hypothesis.

1.1 An example

Suppose we’ve written a run length encoding system and we want to test it out.

We have the following code which I took straight from the Rosetta Code wiki (OK, I removed some commented out code and fixed the formatting, but there are no functional modifications):

```python
def encode(input_string):
    count = 1
    prev = ''
    lst = []
    for character in input_string:
        if character != prev:
            if prev:
                entry = (prev, count)
                lst.append(entry)
                count = 1
            prev = character
        else:
            count += 1
    else:
        entry = (character, count)
    lst.append(entry)
    return lst

def decode(lst):
    q = ''
    for character, count in lst:
        q += character * count
    return q
```

We want to write a test for this that will check some invariant of these functions.

The invariant one tends to try when you’ve got this sort of encoding / decoding is that if you encode something and then decode it then you get the same value back.

Let’s see how you’d do that with Hypothesis:
from hypothesis import given
from hypothesis.strategies import text

given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s

(For this example we’ll just let pytest discover and run the test. We’ll cover other ways you could have run it later).
The text function returns what Hypothesis calls a search strategy. An object with methods that describe how to generate
and simplify certain kinds of values. The @given decorator then takes our test function and turns it into a parametrized
one which, when called, will run the test function over a wide range of matching data from that strategy.

Anyway, this test immediately finds a bug in the code:

Falsifying example: test_decode_inverts_encode(s='')
UnboundLocalError: local variable 'character' referenced before assignment

Hypothesis correctly points out that this code is simply wrong if called on an empty string.
If we fix that by just adding the following code to the beginning of the function then Hypothesis tells us the code is
correct (by doing nothing as you’d expect a passing test to).

if not input_string:
    return []

If we wanted to make sure this example was always checked we could add it in explicitly:

from hypothesis import given, example
from hypothesis.strategies import text
@given(text())
@example('
    def test_decode_inverts_encode(s):
        assert decode(encode(s)) == s

You don’t have to do this, but it can be useful both for clarity purposes and for reliably hitting hard to find examples.
Also in local development Hypothesis will just remember and reuse the examples anyway, but there’s not currently a
very good workflow for sharing those in your CI.

It’s also worth noting that both example and given support keyword arguments as well as positional. The following
would have worked just as well:

@given(s=text())
@example(s='')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s

Anyway, suppose we had a more interesting bug and forgot to reset the count each time.
Hypothesis quickly informs us of the following example:

Falsifying example: test_decode_inverts_encode(s='001')

Note that the example provided is really quite simple. Hypothesis doesn’t just find any counter-example to your tests,
it knows how to simplify the examples it finds to produce small easy to understand ones. In this case, two identical
values are enough to set the count to a number different from one, followed by another distinct value which should
have reset the count but in this case didn’t.
The examples Hypothesis provides are valid Python code you can run. Any arguments that you explicitly provide when calling the function are not generated by Hypothesis, and if you explicitly provide all the arguments Hypothesis will just call the underlying function the once rather than running it multiple times.

## 1.2 Installing

Hypothesis is available on pypi as “hypothesis”. You can install it with:

```
pip install hypothesis
```

or

```
conda install hypothesis
```

If you want to install directly from the source code (e.g. because you want to make changes and install the changed version) you can do this with:

```
python setup.py install
```

You should probably run the tests first to make sure nothing is broken. You can do this with:

```
python setup.py test
```

Note that if they’re not already installed this will try to install the test dependencies.

You may wish to do all of this in a virtualenv. For example:

```
virtualenv venv
source venv/bin/activate
pip install hypothesis
```

Will create an isolated environment for you to try hypothesis out in without affecting your system installed packages.

## 1.3 Running tests

In our example above we just let pytest discover and run our tests, but we could also have run it explicitly ourselves:

```
if __name__ == '__main__':
    test_decode_inverts_encode()
```

We could also have done this as a unittest TestCase:

```
import unittest

class TestEncoding(unittest.TestCase):
    @given(text())
    def test_decode_inverts_encode(self, s):
        self.assertEqual(decode(encode(s)), s)

if __name__ == '__main__':
    unittest.main()
```

A detail: This works because Hypothesis ignores any arguments it hasn’t been told to provide (positional arguments start from the right), so the self argument to the test is simply ignored and works as normal. This also means that Hypothesis will play nicely with other ways of parametrizing tests. e.g it works fine if you use pytest fixtures for some arguments and Hypothesis for others.
1.4 Writing tests

A test in Hypothesis consists of two parts: A function that looks like a normal test in your test framework of choice but with some additional arguments, and a @given decorator that specifies how to provide those arguments.

Here are some other examples of how you could use that:

```python
from hypothesis import given
import hypothesis.strategies as st

given(st.integers(), st.integers())
def test_ints_are_commutative(x, y):
    assert x + y == y + x

@given(x=st.integers(), y=st.integers())
def test_ints_cancel(x, y):
    assert (x + y) - y == x

given(st.lists(st.integers()))
def test_reversing_twice_gives_same_list(xs):
    # This will generate lists of arbitrary length (usually between 0 and 100 elements) whose elements are integers.
    ys = list(xs)
    ys.reverse()
    ys.reverse()
    assert xs == ys

given(st.tuples(st.booleans(), st.text()))
def test_look_tuples_work_too(t):
    # A tuple is generated as the one you provided, with the corresponding types in those positions.
    assert len(t) == 2
    assert isinstance(t[0], bool)
    assert isinstance(t[1], str)
```

Note that as we saw in the above example you can pass arguments to @given either as positional or as keywords.

1.5 Where to start

You should now know enough of the basics to write some tests for your code using Hypothesis. The best way to learn is by doing, so go have a try.

If you’re stuck for ideas for how to use this sort of test for your code, here are some good starting points:

1. Try just calling functions with appropriate random data and see if they crash. You may be surprised how often this works. e.g. note that the first bug we found in the encoding example didn’t even get as far as our assertion: It crashed because it couldn’t handle the data we gave it, not because it did the wrong thing.

2. Look for duplication in your tests. Are there any cases where you’re testing the same thing with multiple different examples? Can you generalise that to a single test using Hypothesis?

3. This piece is designed for an F# implementation, but is still very good advice which you may find helps give you good ideas for using Hypothesis.

If you have any trouble getting started, don’t feel shy about asking for help.
The Hypothesis community is small for the moment but is full of excellent people who can answer your questions and help you out. Please do join us.

The two major places for community discussion are:

- The mailing list.
- An IRC channel: #hypothesis on freenode.

Feel free to use these to ask for help, provide feedback, or discuss anything remotely Hypothesis related at all.

The IRC channel is the more active of the two. If you don’t know how to use IRC, don’t worry about it. Just click here to sign up to IRCCloud and log in (don’t worry, it’s free).

(IRCCloud is made by friends of mine, but that’s not why I’m recommending it. I’m recommending it because it’s great).

### 2.1 Code of conduct

Note that everyone in these spaces is expected to abide by a code of conduct, the Recurse Center social rules. This is an inclusive environment for people from a variety of backgrounds and skill levels. Prejudice and aggression are unwelcome and everyone should be treated with respect.

I’ll do my best to pay attention to peoples' behaviour, but if you see anyone violating these rules and I haven’t noticed, please alert me and I’ll deal with it. Usually I will simply ask people to modify their behaviour, but for particularly severe transgressions, repeat offenders or those unwilling to change their ways I’ll ban them from the community.
The Purpose of Hypothesis

What is Hypothesis for?
From the perspective of a user, the purpose of Hypothesis is to make it easier for you to write better tests.

From my perspective as the author, that is of course also a purpose of Hypothesis, but (if you will permit me to indulge in a touch of megalomania for a moment), the larger purpose of Hypothesis is to drag the world kicking and screaming into a new and terrifying age of high quality software.

Software is, as they say, eating the world. Software is also terrible. It’s buggy, insecure and generally poorly thought out. This combination is clearly a recipe for disaster.

And the state of software testing is even worse. Although it’s fairly uncontroversial at this point that you should be testing your code, can you really say with a straight face that most projects you’ve worked on are adequately tested?

A lot of the problem here is that it’s too hard to write good tests. Your tests encode exactly the same assumptions and fallacies that you had when you wrote the code, so they miss exactly the same bugs that you missed when you wrote the code.

Meanwhile, there are all sorts of tools for making testing better that are basically unused. The original Quickcheck is from 1999 and the majority of developers have not even heard of it, let alone used it. There are a bunch of half-baked implementations for most languages, but very few of them are worth using.

The goal of Hypothesis is to bring advanced testing techniques to the masses, and to provide an implementation that is so high quality that it is easier to use than it is not to use them. Where I can, I will beg, borrow and steal every good idea I can find that someone has had to make software testing better. Where I can’t, I will invent new ones.

Quickcheck is the start, but I also plan to integrate ideas from fuzz testing (a planned future feature is to use coverage information to drive example selection, and the example saving database is already inspired by the workflows people use for fuzz testing), and am open to and actively seeking out other suggestions and ideas.

The plan is to treat the social problem of people not using these ideas as a bug to which there is a technical solution: Does property-based testing not match your workflow? That’s a bug, lets fix it by figuring out how to integrate Hypothesis into it. Too hard to generate custom data for your application? That’s a bug. Lets fix it by figuring out how to make it easier, or how to take something you’re already using to specify your data and derive a generator from that automatically. Find the explanations of these advanced ideas hopelessly obtuse and hard to follow? That’s a bug. Let’s provide you with an easy API that lets you test your code better without a PhD in software verification.

Grand ambitions, I know, and I expect ultimately the reality will be somewhat less grand, but so far in about three months of development, Hypothesis has become the most solid implementation of Quickcheck ever seen in a mainstream language (as long as we don’t count Scala as mainstream yet), and at the same time managed to significantly push forward the state of the art, so I think there’s reason to be optimistic.
This is an account of slightly less common Hypothesis features that you don’t need to get started but will nevertheless make your life easier.

### 4.1 Making assumptions

Sometimes hypothesis doesn’t give you exactly the right sort of data you want - it’s mostly of the right shape, but some examples won’t work and you don’t want to care about them. You *can* just ignore these by aborting the test early, but this runs the risk of accidentally testing a lot less than you think you are. Also it would be nice to spend less time on bad examples - if you’re running 200 examples per test (the default) and it turns out 150 of those examples don’t match your needs, that’s a lot of wasted time.

The way Hypothesis handles this is to let you specify things which you *assume* to be true. This lets you abort a test in a way that marks the example as bad rather than failing the test. Hypothesis will use this information to try to avoid similar examples in future.

For example suppose had the following test:

```python
from hypothesis import given
from hypothesis.strategies import floats

given(floats())
def test_negation_is_self_inverse(x):
    assert x == -(-x)
```

Running this gives us:

```
Falsifying example: test_negation_is_self_inverse(x=float('nan')) AssertionError
```

This is annoying. We know about NaN and don’t really care about it, but as soon as Hypothesis finds a NaN example it will get distracted by that and tell us about it. Also the test will fail and we want it to pass.

So lets block off this particular example:

```python
from hypothesis import given, assume
from hypothesis.strategies import floats
from math import isnan

given(floats())
def test_negation_is_self_inverse_for_non_nan(x):
    assume(not isnan(x))
    assert x == -(-x)
```
And this passes without a problem.

`assume` throws an exception which terminates the test when provided with a false argument. It’s essentially an `assert`, except that the exception it throws is one that Hypothesis identifies as meaning that this is a bad example, not a failing test.

In order to avoid the easy trap where you assume a lot more than you intended, Hypothesis will fail a test when it can’t find enough examples passing the assumption.

If we’d written:

```python
from hypothesis import given, assume
from hypothesis.strategies import floats

@given(floats())
def test_negation_is_self_inverse_for_non_nan(x):
    assume(False)
    assert x == -(-x)
```

Then on running we’d got the exception:

```
Unsatisfiable: Unable to satisfy assumptions of hypothesis test_negation_is_self_inverse_for_non_nan.
Only 0 examples found after 0.0791318 seconds
```

### 4.1.1 How good is `assume`?

Hypothesis has an adaptive exploration strategy to try to avoid things which falsify assumptions, which should generally result in it still being able to find examples in hard to find situations.

Suppose we had the following:

```python
@given(lists(integers()))
def test_sum_is_positive(xs):
    assert sum(xs) > 0
```

Unsurprisingly this fails and gives the falsifying example `[]`.

Adding `assume(xs)` to this removes the trivial empty example and gives us `[0]`.

Adding `assume(all(x > 0 for x in xs))` and it passes: A sum of a list of positive integers is positive.

The reason that this should be surprising is not that it doesn’t find a counter-example, but that it finds enough examples at all.

In order to make sure something interesting is happening, suppose we wanted to try this for long lists. e.g. suppose we added an `assume(len(xs) > 10)` to it. This should basically never find an example: A naive strategy would find fewer than one in a thousand examples, because if each element of the list is negative with probability half, you’d have to have ten of these go the right way by chance. In the default configuration Hypothesis gives up long before it’s tried 1000 examples (by default it tries 200).

Here’s what happens if we try to run this:

```python
@given(lists(integers()))
def test_sum_is_positive(xs):
    assume(len(xs) > 10)
    assume(all(x > 0 for x in xs))
    print(xs)
    assert sum(xs) > 0

In: test_sum_is_positive()
[17, 12, 7, 13, 11, 3, 6, 9, 8, 11, 47, 27, 1, 31, 1]
```
As you can see, Hypothesis doesn’t find many examples here, but it finds some - enough to keep it happy.

In general if you can shape your strategies better to your tests you should - for example integers_in_range(1, 1000) is a lot better than assume(1 <= x <= 1000), but assume will take you a long way if you can’t.

### 4.2 Settings

Hypothesis tries to have good defaults for its behaviour, but sometimes that’s not enough and you need to tweak it.

The mechanism for doing this is the Settings object. You can pass this to a @given invocation as follows:

```python
from hypothesis import given, Settings

given(integers(), settings=Settings(max_examples=500))
def test_this_thoroughly(x):
    pass
```

This uses a Settings object which causes the test to receive a much larger set of examples than normal.

There is a Settings.default object. This is both a Settings object you can use, but additionally any changes to the default object will be picked up as the defaults for newly created settings objects.

```bash
>>> from hypothesis import Settings
>>> s = Settings()
>>> s.max_examples
200
>>> Settings.default.max_examples = 100
>>> t = Settings()
>>> t.max_examples
100
>>> s.max_examples
200
```

You can also override the default locally by using a settings object as a context manager:

```bash
>>> with Settings(max_examples=150):
...     print(Settings().max_examples)
... 150
>>> Settings().max_examples
200
```

Note that after the block exits the default is returned to normal.

You can use this by nesting test definitions inside the context:

```python
from hypothesis import given, Settings

with Settings(max_examples=500):
    given(integers())
def test_this_thoroughly(x):
    pass
```
All Settings objects created or tests defined inside the block will inherit their defaults from the settings object used as the context. You can still override them with custom defined settings of course.

As well as max_examples there are a variety of other settings you can use. help(Settings) in an interactive environment will give you a full list of them.

### 4.2.1 Seeing intermediate result

To see what’s going on while Hypothesis runs your tests, you can turn up the verbosity setting. This works with both find and @given.

(The following examples are somewhat manually truncated because the results of verbose output are, well, verbose, but they should convey the idea).

```python
>>> from hypothesis import find, Settings, Verbosity
>>> from hypothesis.strategies import lists, booleans
>>> find(lists(booleans()), any, settings=Settings(verbosity=Verbosity.verbose))
Found satisfying example [True, True, ...
Shrunk example to [False, False, False, True, ...
Shrunk example to [False, False, True, False, False, ...
Shrunk example to [False, True, False, True, True, ...
Shrunk example to [True, True, True]
Shrunk example to [True, True]
Shrunk example to [True]
[True]

>>> from hypothesis import given
>>> from hypothesis.strategies import integers()
>>> Settings.default.verbosity = Verbosity.verbose
>>> @given(integers())
... def test_foo(x):
...     assert x > 0
...
>>> test_foo()
Trying example: test_foo(x=-565872324465712963891750807252490657219)
Traceback (most recent call last):
...
File "<stdin>", line 3, in test_foo
AssertionError

Trying example: test_foo(x=565872324465712963891750807252490657219)
Trying example: test_foo(x=0)
Traceback (most recent call last):
...
File "<stdin>", line 3, in test_foo
AssertionError
Falsifying example: test_foo(x=0)
Traceback (most recent call last):
...
AssertionError
```

The four levels are quiet, normal, verbose and debug. normal is the default, while in quiet Hypothesis will not print anything out, even the final falsifying example. debug is basically verbose but a bit more so. You probably don’t want it.

You can also override the default by setting the environment variable HYPOTHESIS_VERBOSITY_LEVEL to the name of the level you want. So e.g. setting HYPOTHESIS_VERBOSITY_LEVEL=verbose will run all your tests printing intermediate results and errors.
4.3 Defining strategies

The type of object that is used to explore the examples given to your test function is called a SearchStrategy. These are created using the functions exposed in the hypothesis.strategies module.

Many of these strategies expose a variety of arguments you can use to customize generation. For example for integers you can specify min and max values of integers you want:

```python
>>> from hypothesis.strategies import integers,
RandomGeometricIntStrategy(), WideRangeIntStrategy()
>>> integers(min_value=0)
IntegersFromStrategy(0)
>>> integers(min_value=0, max_value=10)
BoundedIntStrategy(0, 10)
```

If you want to see exactly what a strategy produces you can ask for an example:

```python
>>> integers(min_value=0, max_value=10).example()
7
```

Many strategies are build out of other strategies. For example, if you want to define a tuple you need to say what goes in each element:

```python
>>> from hypothesis.strategies import tuples
>>> tuples(integers(), integers()).example()
(-1953, 85733644253897814191482551773726674360154905303788466954)
```

Further details are available in a separate document.

4.4 The gory details of given parameters

The `@given` decorator may be used to specify what arguments of a function should be parametrized over. You can use either positional or keyword arguments or a mixture of the two.

For example all of the following are valid uses:

```python
@given(integers(), integers())
def a(x, y):
    pass

@given(integers())
def b(x, y):
    pass

@given(y=integers())
def c(x, y):
    pass

@given(x=integers(), y=integers())
def d(x, **kwargs):
    pass

class SomeTest(TestCase):
    @given(integers())
```
The following are not:

```python
@given(integers(), integers(), integers())
def e(x, y):
    pass
@given(x=integers())
def f(x, y):
    pass
@given()
def f(x, y):
    pass
```

The rules for determining what are valid uses of `given` are as follows:

1. Arguments passed as keyword arguments must cover the right hand side of the argument list. That is, if you provide an argument as a keyword you must also provide everything to the right of it.

2. Positional arguments fill up from the right, starting from the first argument not covered by a keyword argument. (Note: Mixing keyword and positional arguments is supported but deprecated as its semantics are highly confusing and difficult to support. You’ll get a warning if you do).

3. If the function has variable keywords, additional arguments will be added corresponding to any keyword arguments passed. These will be to the right of the normal argument list in an arbitrary order.

4. varargs are forbidden on functions used with `@given`.

If you don’t have kwargs then the function returned by `@given` will have the same argspec (i.e. same arguments, keyword arguments, etc) as the original but with different defaults.

The reason for the “filling up from the right” behaviour is so that using `@given` with instance methods works: `self` will be passed to the function as normal and not be parametrized over.

### 4.5 Custom function execution

Hypothesis provides you with a hook that lets you control how it runs examples.

This lets you do things like set up and tear down around each example, run examples in a subprocess, transform coroutine tests into normal tests, etc.

The way this works is by introducing the concept of an executor. An executor is essentially a function that takes a block of code and run it. The default executor is:

```python
def default_executor(function):
    return function()
```

You define executors by defining a method `execute_example` on a class. Any test methods on that class with `@given` used on them will use `self.execute_example` as an executor with which to run tests. For example, the following executor runs all its code twice:

```python
from unittest import TestCase
class TestTryReallyHard(TestCase):
    @given(integers())
    def test_something(self, i):
```
perform_some_unreliable_operation(i)

```python
def execute_example(self, f):
f()
    return f()
```

Note: The functions you use in map, etc. will run inside the executor; i.e. they will not be called until you invoke the function passed to setup_example.

Methods of a BasicStrategy however will typically be called whenever. This may happen inside your executor or outside. This is why they have a “Warning you have no control over the lifecycle of these values” attached.

### 4.5.1 Fork before each test

An obstacle you can run into if you want to use Hypothesis to test native code is that your C code segfaults, or fails a C level assertion, and it causes the whole process to exit hard and Hypothesis just cries a little and doesn’t know what is going on, so can’t minimize an example for you.

The solution to this is to run your tests in a subprocess. The process can die as messily as it likes and Hypothesis will be sitting happily in the controlling process unaffected by the crash. Hypothesis provides a custom executor for this:

```python
from hypothesis.testrunners.forking import ForkingTestCase

class TestForking(ForkingTestCase):
    @given(integers())
    def test_handles_abnormal_exit(self, i):
        os._exit(1)

    @given(integers())
    def test_normal_exceptions_work_too(self, i):
        assert False
```

Exceptions that occur in the child process will be seamlessly passed back to the parent. Abnormal exits that do not throw an exception in the child process will be turned into an AbnormalExit exception.

There are currently some limitations to this approach:

1. Exceptions which are not pickleable will be turned into abormal exits.
2. Tracebacks from exceptions are not properly recreated in the parent process.
3. Code called in the child process will not be recorded by coverage.
4. This is only supported on platforms with os.fork. e.g. it will not work on Windows.

Some of these limitations should be resolvable in time.

### 4.6 Using Hypothesis to find values

You can use Hypothesis’s data exploration features to find values satisfying some predicate:

```python
>>> from hypothesis import find
>>> from hypothesis.strategies import sets, lists, integers
>>> find(lists(integers()), lambda x: sum(x) >= 10)
[10]
>>> find(lists(integers()), lambda x: sum(x) >= 10 and len(x) >= 3)
[0, 0, 10]
```
The first argument to find describes data in the usual way for an argument to given, and supports all the same data types. The second is a predicate it must satisfy.

Of course not all conditions are satisfiable. If you ask Hypothesis for an example to a condition that is always false it will raise an error:

```python
>>> find(integers(), lambda x: False)
Traceback (most recent call last):
... hypothesis.errors.NoSuchExample: No examples of condition lambda x: <unknown>
>>> from hypothesis.strategies import booleans
>>> find(booleans(), lambda x: False)
Traceback (most recent call last):
... hypothesis.errors.DefinitelyNoSuchExample: No examples of condition lambda x: <unknown> (all 2 considered)
```

(The “lambda x: unknown” is because Hypothesis can’t retrieve the source code of lambdas from the interactive python console. It gives a better error message most of the time which contains the actual condition)

The reason for the two different types of errors is that there are only a small number of booleans, so it is feasible for Hypothesis to enumerate all of them and simply check that your condition is never true.

## 4.7 Providing explicit examples

You can explicitly ask Hypothesis to try a particular example as follows:

```python
from hypothesis import given, example
from hypothesis.strategies import text

given(text())
@example("Hello world")
@example(x="Some very long string")
def test_some_code(x):
    assert True
```

Hypothesis will run all examples you’ve asked for first. If any of them fail it will not go on to look for more examples.

This can be useful both because it’s easier to share and version examples in source code than it is to share the example database, and it can also allow you to feed specific examples that Hypothesis is unlikely to figure out on its own.

It doesn’t matter whether you put the example decorator before or after given. Any permutation of the decorators in the above will do the same thing.

Note that examples can be positional or keyword based. If they’re positional then they will be filled in from the right when calling, so things like the following will also work:

```python
from unittest import TestCase
from hypothesis import given, example
from hypothesis.strategies import text

class TestThings(TestCase):
    @given(text())
    @example("Hello world")
    @example(x="Some very long string")
```
It is not permitted for a single example to be a mix of positional and keyword arguments. Either are fine, and you can use one in one example and the other in another example if for some reason you really want to, but a single example must be consistent.
What you can generate and how

The general philosophy of Hypothesis data generation is that everything should be possible to generate and most things should be easy. Most things in the standard library is more aspirational than achieved, the state of the art is already pretty good.

This document is a guide to what strategies are available for generating data and how to build them. Strategies have a variety of other important internal features, such as how they simplify, but the data they can generate is the only public part of their API.

Functions for building strategies are all available in the hypothesis.strategies module. The salient functions from it are as follows:

- `hypothesis.strategies.just(value)`: Return a strategy which only generates value.
  
  Note: value is not copied. Be wary of using mutable values.

- `hypothesis.strategies.one_of(arg, *args)`: Return a strategy which generates values from any of the argument strategies.

- `hypothesis.strategies.integers(min_value=None, max_value=None)`: Returns a strategy which generates integers (in Python 2 these may be ints or longs).
  
  If min_value is not None then all values will be >= min_value. If max_value is not None then all values will be <= max_value.

- `hypothesis.strategies.booleans()`: Returns a strategy which generates instances of bool.

- `hypothesis.strategies.floats(min_value=None, max_value=None)`: Returns a strategy which generates floats. If min_value is not None, all values will be >= min_value. If max_value is not None, all values will be <= max_value.
  
  Where not explicitly ruled out by the bounds, all of infinity, -infinity and NaN are possible values generated by this strategy.

- `hypothesis.strategies.complex_numbers()`: Returns a strategy that generates complex numbers.

- `hypothesis.strategies.tuples(*args)`: Return a strategy which generates a tuple of the same length as args by generating the value at index i from args[i].
  
  e.g. tuples(integers(), integers()) would generate a tuple of length two with both values an integer.

- `hypothesis.strategies.lists(elements=None, min_size=None, average_size=None, max_size=None)`: Returns a list containing values drawn from elements length in the interval [min_size, max_size] (no bounds
in that direction if these are None). If max_size is 0 then elements may be None and only the empty list will be drawn.

average_size may be used as a size hint to roughly control the size of list but it may not be the actual average of sizes you get, due to a variety of factors.

```python
hypothesis.strategies.sets(elements=None, min_size=None, average_size=None, max_size=None)
```

This has the same behaviour as lists, but returns sets instead.

Note that Hypothesis cannot tell if values are drawn from elements are hashable until running the test, so you can define a strategy for sets of an unhashable type but it will fail at test time.

```python
hypothesis.strategies.frozensets(elements=None, min_size=None, average_size=None, max_size=None)
```

This is identical to the sets function but instead returns frozensets.

```python
hypothesis.strategies.fixed_dictionaries(mapping)
```

Generate a dictionary of the same type as mapping with a fixed set of keys mapping to strategies. mapping must be a dict subclass.

Generated values have all keys present in mapping, with the corresponding values drawn from mapping[key]. If mapping is an instance of OrderedDict the keys will also be in the same order, otherwise the order is arbitrary.

```python
hypothesis.strategies.dictionaries(keys, values, dict_class=<type 'dict'>, min_size=None, average_size=None, max_size=None)
```

Generates dictionaries of type dict_class with keys drawn from the keys argument and values drawn from the values argument.

The size parameters have the same interpretation as for lists.

```python
hypothesis.strategies.streaming(elements)
```

Generates an infinite stream of values where each value is drawn from elements.

The result is iterable (the iterator will never terminate) and indexable.

```python
hypothesis.strategies.text(alphabet=None, min_size=None, average_size=None, max_size=None)
```

Generates values of a unicode text type (unicode on python 2, str on python 3) with values drawn from alphabet, which should be an iterable of length one strings or a strategy generating such. If it is None it will default to generating the full unicode range. If it is an empty collection this will only generate empty strings.

min_size, max_size and average_size have the usual interpretations.

```python
hypothesis.strategies.binary(min_size=None, average_size=None, max_size=None)
```

Generates the appropriate binary type (str in python 2, binary in python 3).

min_size, average_size and max_size have the usual interpretations.

```python
hypothesis.strategies.basic(basic=None, generate_parameter=None, generate=None, simplify=None, copy=None)
```

Provides a facility to write your own strategies with significantly less work.

See documentation for more details.

```python
hypothesis.strategies.fractions()
```

Generates instances of fractions.Fraction.

```python
hypothesis.strategies.decimals()
```

Generates instances of decimals.Decimal.

```python
hypothesis.strategies.builds(target, *args, **kwargs)
```

Generates values by drawing from args and kwargs and passing them to target in the appropriate argument position.
e.g. builds(target, integers(), flag=booleans()) would draw an integer i and a boolean b and call target(i, flag=b).

## 5.1 Infinite streams

Sometimes you need examples of a particular type to keep your test going but you’re not sure how many you’ll need in advance. For this, we have streaming types.

```python
>>> from hypothesis import strategy
>>> from hypothesis.strategies import streaming, integers

>>> x = strategy(streaming(integers())).example()
>>> x
Stream(...)
>>> x[2]
209
>>> x
Stream(32, 132, 209, ...)
>>> x[10]
130
>>> x
Stream(32, 132, 209, 843, -19, 58, 141, -1046, 37, 243, 130, ...)
```

Think of a Stream as an infinite list where we’ve only evaluated as much as we need to. As per above, you can index into it and the stream will be evaluated up to that index and no further.

You can iterate over it too (warning: iter on a stream given to you by Hypothesis in this way will never terminate):

```python
>>> it = iter(x)
>>> next(it)
32
>>> next(it)
132
>>> next(it)
209
>>> next(it)
843
```

Slicing will also work, and will give you back Streams. If you set an upper bound then iter on those streams will terminate:

```python
>>> list(x[:5])
[32, 132, 209, 843, -19]
>>> y = x[1::2]
>>> y
Stream(...)
>>> y[0]
132
>>> y[1]
843
>>> y
Stream(132, 843, ...)
```

You can also apply a function to transform a stream:

```python
>>> t = strategy(streaming(int)).example()
>>> tm = t.map(lambda n: n * 2)
>>> tm[0]
26
>>> t[0]
```

### 5.1.1 Infinite streams
map creates a new stream where each element of the stream is the function applied to the corresponding element of
the original stream. Evaluating the new stream will force evaluating the original stream up to that index.
(Warning: This isn’t the map builtin. In Python 3 the builtin map should do more or less the right thing, but in Python
2 it will never terminate and will just eat up all your memory as it tries to build an infinitely long list)
These are the only operations a Stream supports. There are a few more internal ones, but you shouldn’t rely on them.

5.1.1 Adapting strategies

Often it is the case that a strategy doesn’t produce exactly what you want it to and you need to adapt it. Sometimes
you can do this in the test, but this hurts reuse because you then have to repeat the adaption in every test.
Hypothesis gives you ways to build strategies from other strategies given functions for transforming the data.

5.2 Mapping

Map is probably the easiest and most useful of these to use. If you have a strategy s and a function f, then an example
s.map(f).example() is f(s.example()). i.e. we draw an example from s and then apply f to it.

e.g.:

```python
>>> strategy([int]).map(sorted).example()
[1, 5, 17, 21, 24, 30, 45, 82, 88, 88, 90, 96, 105]
```

Note that many things that you might use mapping for can also be done with the builds function in hypothe-
sis.strategies.

5.3 Filtering

filter lets you reject some examples. s.filter(f).example() is some example of s such that f(s) is truthy.

```python
>>> strategy(int).filter(lambda x: x > 11).example()
1873
>>> strategy(int).filter(lambda x: x > 11).example()
73
```

It’s important to note that filter isn’t magic and if your condition is too hard to satisfy then this can fail:

```python
>>> strategy(int).filter(lambda x: False).example()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/david/projects/hypothesis/src/hypothesis/searchstrategy/strategies.py", line 175, in example
    'Could not find any valid examples in 20 tries'
  hypothesis.errors>NoExamples: Could not find any valid examples in 20 tries
```

In general you should try to use filter only to avoid corner cases that you don’t want rather than attempting to cut out
a large chunk of the search space.
A technique that often works well here is to use map to first transform the data and then use filter to remove things that didn’t work out. So for example if you wanted pairs of integers \((x,y)\) such that \(x < y\) you could do the following:

```python
>>> strategy((int, int)).map(
... lambda x: tuple(sorted(x))).filter(lambda x: x[0] != x[1]).example()
(42, 1281698)
```

## 5.4 Chaining strategies together

Finally there is flatmap. Flatmap draws an example, then turns that example into a strategy, then draws an example from that strategy.

It may not be obvious why you want this at first, but it turns out to be quite useful because it lets you generate different types of data with relationships to each other.

For example suppose we wanted to generate a list of lists of the same length:

```python
>>> from hypothesis.strategies import integers, lists
>>> from hypothesis import find
>>> rectangle_lists = integers(min_value=0, max_value=10).flatmap(lambda n:
... lists(lists(integers(), min_size=n, max_size=n)))
>>> find(rectangle_lists, lambda x: True)
[]
>>> find(rectangle_lists, lambda x: len(x) >= 10)
[[], [], [], [], [], [], [], [], [], []]
>>> find(rectangle_lists, lambda t: len(t) >= 3 and len(t[0]) >= 3)
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> find(rectangle_lists, lambda t: sum(len(s) for s in t) >= 10)
[[0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]
```

In this example we first choose a length for our tuples, then we build a strategy which generates lists containing lists precisely of that length. The finds show what simple examples for this look like.

Most of the time you probably don’t want flatmap, but unlike filter and map which are just conveniences for things you could just do in your tests, flatmap allows genuinely new data generation that you wouldn’t otherwise be able to easily do.

(If you know Haskell: Yes, this is more or less a monadic bind. If you don’t know Haskell, ignore everything in these parentheses. You do not need to understand anything about monads to use this, or anything else in Hypothesis).

### 5.4.1 Defining entirely new strategies

The full SearchStrategy API is only “semi-public”, in that it may (but usually won’t) break between minor versions but won’t break between patch releases.

However Hypothesis exposes a simplified version of the interface that you can use to build pretty good strategies. In general it’s pretty strongly recommended that you don’t use this if you can build your strategy out of existing ones, but it works perfectly well.

Here is an example of using the simplified interface:

```python
from hypothesis.searchstrategy import BasicStrategy

class Bitfields(BasicStrategy):
    """A BasicStrategy for generating 128 bit integers to be treated as if they
```
were bitfields."

```python
def generate_parameter(self, random):
    # This controls the shape of the data that can be generated by
    # randomly screening off some bits.
    return random.getrandbits(128)

def generate(self, random, parameter_value):
    # This generates a random value subject to a parameter we have
    # previously generated
    return parameter_value & random.getrandbits(128)

def simplify(self, random, value):
    # Simplify by settings bits to zero.
    for i in range(128):
        k = 1 << i
        # It’s important to test this because otherwise it would create a
        # cycle where value simplifies to value. This would cause
        # Hypothesis to get stuck on that value and not be able to simplify
        # it further.
        if value & k:
            yield value & (~k)

def copy(self, value):
    # integers are immutable so there’s no need to copy them
    return value
```

Only generate is strictly necessary to implement. copy will default to using deepcopy, generate_parameter will default
to returning None, and simplify will default to not simplifying.

The reason why the parameters are important is that they let you “shape” the data so that it works with adaptive
assumptions, which work by being more likely to reuse parameter values that don’t cause assumptions to be violated.

Simplify is of course what Hypothesis uses to produce simpler examples. It will greedily apply it to your data to
produce the simplest example it possible can. You should avoid having cycles or unbounded paths in the graph, as this
will tend to hurt example quality and performance.

Instances of BasicStrategy are not actually strategies and must be converted to them using the basic function from
hypothesis.strategies. You can convert either a class or an instance:

```python
>>> basic(Bitfields).example()
704938930150216502654673882738917538
>>> strategy(Bitfields()).example()
180947746395888412520415493036267606532
```

You can also skip the class definition if you prefer and just pass functions to basic. e.g.

```python
>>> basic(generate=lambda random, _: random.getrandbits(8)).example()
88
```

The arguments to basic have the same names as the methods you would define on BasicStrategy.

Caveats:

- Remember that BasicStrategy is not a subclass of SearchStrategy, only convertible to one.

- The values produced by BasicStrategy are opaque to Hypothesis in a way that ones it is more intimately familiar
  with are not, because it’s impossible to safely and sensibly deduplicate arbitrary Python objects. This is mostly
  fine but it blocks certain heuristics and optimisations Hypothesis uses for improving the simplification process.
  As such implementations using BasicStrategy might get slightly worse examples than the equivalent native ones.
• You should not use BasicData for anything which you need control over the life cycle of, e.g. ORM objects. Hypothesis will keep instances of these values around for a potentially arbitrarily long time and will not do any clean up for disposing of them other than letting them be GCed as normal.

However if it’s genuinely the best way for you to do it, you should feel free to use BasicStrategy. These caveats should be read in the light of the fact that the full Hypothesis SearchStrategy interface is really very powerful, and the ones using BasicStrategy are merely a bit better than the normal quickcheck interface.

### 5.4.2 Using the SearchStrategy API directly

If you’re really super enthused about this search strategies thing and you want to learn all the gory details of how it works under the hood, you can use the full blown raw SearchStrategy interface to experience the full power of Hypothesis.

This is only semi-public API, meaning that it may break between minor versions but will not break in patch versions, but it should be considered relatively stable and most minor versions won’t break it.

```python
class hypothesis.strategies.SearchStrategy

A SearchStrategy is an object that knows how to explore data of a given type.

Except where noted otherwise, methods on this class are not part of the public API and their behaviour may change significantly between minor version releases. They will generally be stable between patch releases.

With that in mind, here is how SearchStrategy works.

A search strategy is responsible for generating, simplifying and serializing examples for saving.

In order to do this a strategy has three types (where type here is more precise than just the class of the value. For example a tuple of ints should be considered different from a tuple of strings):

1. The strategy parameter type
2. The strategy template type
3. The generated type

Of these, the first two should be considered to be private implementation details of a strategy and the only valid thing to do them is to pass them back to the search strategy. Additionally, templates may be compared for equality and hashed.

Templates must be of quite a restricted type. A template may be any of the following:

1. Any instance of the types bool, float, int, str (unicode on 2.7)
2. None
3. Any tuple or namedtuple of valid template types
4. Any frozenset of valid template types

This may be relaxed a bit in future, but the requirement that templates are hashable probably won’t be.

This may all seem overly complicated but it’s for a fairly good reason. For more discussion of the motivation see [http://hypothesis.readthedocs.org/en/master/internals.html](http://hypothesis.readthedocs.org/en/master/internals.html)

Given these, data generation happens in three phases:

1. Draw a parameter value from a random number (defined by draw_parameter)
2. Given a parameter value and a Random, draw a random template
3. Reify a template value, deterministically turning it into a value of the desired type.

Data simplification proceeds on template values, taking a template and providing a generator over some examples of similar but simpler templates.

### 5.4. Chaining strategies together
example()
    Provide an example of the sort of value that this strategy generates. This is biased to be slightly simpler than is typical for values from this strategy, for clarity purposes.

    This method shouldn’t be taken too seriously. It’s here for interactive exploration of the API, not for any sort of real testing.

    This method is part of the public API.

map (pack)
    Returns a new strategy that generates values by generating a value from this strategy and then calling pack() on the result, giving that.

    This method is part of the public API.

flatMap (expand)
    Returns a new strategy that generates values by generating a value from this strategy, say x, then generating a value from strategy(expand(x))

    This method is part of the public API.

filter (condition)
    Returns a new strategy that generates values from this strategy which satisfy the provided condition. Note that if the condition is too hard to satisfy this might result in your tests failing with Unsatisfiable.

    This method is part of the public API.

draw_parameter (random)
    Produce a random valid parameter for this strategy, using only data from the provided random number generator.

draw_template (random, parameter_value)
    Given this Random and this parameter value, produce a random valid template for this strategy.

reify (template)
    Given a template value, deterministically convert it into a value of the desired final type.

to_basic (template)
    Convert a template value for this strategy into basic data.

    Basic data is any of:

        1. A bool, None, an int that fits into 64 bits, or a unicode string
        2. A list of basic data

from_basic (value)
    Convert basic data back to a template, raising BadData if the provided data cannot be converted into a valid template for this strategy.

    It is not required that from_basic(to_basic(template)) == template. It is however required that to_basic(from_basic(data)) == data (if this does not raise an exception).

template_upper_bound = inf
    Provide an upper bound on the number of available templates. The intended interpretation is that template_upper_bound means “if you’ve only found this many templates don’t worry about it”. It is also used internally in a few places for certain optimisations. Generally speaking once this reaches numbers $\geq 2^{32}$ or so you might as well just return float(‘inf’). Note that there may be more distinct templates than there are representable values, because some templates may not reify and some may lead to the same value.

strictly_simpler (x, y)
    Is the left hand argument strictly simpler than the right hand side.

    Required properties:
1. not strictly_simpler(x, y)  
2. not (strictly_simpler(x, y) and strictly_simpler(y, x))  
3. not (strictly_simpler(x, y) and strictly_simpler(y, z) and strictly_simpler(z, x))

This is used for hinting in certain cases. The default implementation of it always returns False and this is perfectly acceptable to leave as is.

**simplifiers** *(random, template)*

Yield a sequence of functions which each take a Random object and a single template and produce a generator over “simpler” versions of that template.

The only other required invariant that each simplifier must satisfy is it should not be the case that strictly_simpler(x, y) for any y in simplify(random, x). That is, it’s OK if the simplify doesn’t produce a strictly simpler value but it must not produce a strictly more complex one.

General tips for a good simplify function:

1. The generator shouldn’t yield too many values. A few hundred is fine, but if you’re generating millions of simplifications you may wish to reconsider your life choices and evaluate which ones actually matter to you.
2. Cycles in simplify are fine, but the simplify graph should be bounded in the sense that there should be no infinite acyclic paths where a1 simplifies to a2 simplifies to ...
3. Try major simplifications first to see if you get lucky. Yield a minimal element, throw out half of your data, etc. Providing shortcuts in the graph will speed up the simplification process a lot.

The template argument is provided to allow picking simplifiers that are likely to be useful. It should be considered only a hint, and each simplifier must be valid (in the sense of not erroring. It doesn’t have to do anything useful) for all templates valid for this strategy.

By default this just yields the basic_simplify function (which in turn by default does not do anything useful). If you override this function and also override basic_simplify you should make sure to yield it, or it will not be called.

**full_simplify** *(random, template)*

A convenience method.

Run each simplifier over this template and yield the results in turn.

The order in which simplifiers are run is lightly randomized from the order in which simplifiers provides them, in order to avoid certain pathological cases.

**basic_simplify** *(random, template)*

A convenience method for subclasses that do not have complex simplification requirements to override.

See simplifiers for details.
Hypothesis has a zero dependency policy for the core library. For things which need a dependency to work, these are
farmed off into additional packages on pypi. These end up putting any additional things you need to import (if there
are any) under the hypothesis.extra namespace.

Generally these will be for providing new sources of data for Hypothesis, or for better integrating it into an existing
testing framework.

### 6.1 hypothesis-datetime

As might be expected, this provides a strategy which generates instances of datetime. It depends on pytz.

hypothesis-datetime lives in the hypothesis.extra.datetime package:

```python
>>> from datetime import datetime
datetimes().example()
datetime.datetime(1705, 1, 20, 0, 32, 0, 973139, tzinfo=<DstTzInfo 'Israel...
```

As you can see, it produces years from quite a wide range. If you want to narrow it down you can ask for a more
specific range of years:

```python
>>> datetimes(min_year=2001, max_year=2010).example()
datetime.datetime(2010, 7, 7, 0, 15, 0, 614034, tzinfo=<DstTzInfo 'Pacif...
```

You can also specify timezones:

```python
>>> import pytz
>> pytz.all_timezones[:3]
['Africa/Abidjan', 'Africa/Accra', 'Africa/Addis_Ababa']
```

If the set of timezones is empty you will get a naive datetime:

```python
>>> datetimes(timezones=pytz.all_timezones[:3]).example()
datetime.datetime(6257, 8, 21, 13, 6, 24, 8751, tzinfo=<DstTzInfo 'Africa/Accra' GMT0:00:00 STD>)
```
You can also explicitly get a mix of naive and non-naive datetimes if you want:

```python
>>> datetimes(allow_naive=True).example()
datetime.datetime(2433, 3, 20, 0, 0, 44, 460383, tzinfo=<DstTzInfo 'Asia/Hovd' HOVT+7:00:00 STD>)
```

### 6.2 hypothesis-fakefactory

Fake-factory is another Python library for data generation. hypothesis-fakefactory is a package which lets you use fake-factory generators to parametrize tests.

It currently only supports the 0.4.2 release of fake-factory, due to some issues with the 0.5.0 release. These are known to be fixed in master but there hasn’t been a release containing the fixes yet.

hypothesis.extra.fakefactory defines a function `fake_factory` which returns a strategy for producing text data from any FakeFactory provider.

So for example the following will parametrize a test by an email address:

```python
>>> fake_factory('email').example()
'tnader@prosacco.info'
```

You can explicitly specify the locale (otherwise it uses any of the available locales), either as a single locale or as several:

```python
>>> fake_factory('name', locale='en_GB').example()
'Antione Gerlach'
```

```python
>>> fake_factory('name', locales=['en_GB', 'cs_CZ']).example()
'Miloš Št’astný'
>>> fake_factory('name', locales=['en_GB', 'cs_CZ']).example()
'Harm Sanford'
```

If you want to your own FakeFactory providers you can do that too, passing them in as a providers argument:

```python
>>> from faker.providers import BaseProvider
>>> class KittenProvider(BaseProvider):
...     def meows(self):
...         return 'meow %d' % (self.random_number(digits=10),)
...
>>> fake_factory('meows', providers=[KittenProvider]).example()
'meow 9139348419'
```

Generally you probably shouldn’t do this unless you’re reusing a provider you already have - Hypothesis’s facilities for strategy generation are much more powerful and easier to use. Consider using something like BasicStrategy instead if you want to write a strategy from scratch. This is only here to provide easy reuse of things you already have.
6.3 hypothesis-pytest

hypothesis-pytest is the world’s most basic pytest plugin. Install it to get slightly better integrated example reporting when using @given and running under pytest. That’s basically all it does.

6.4 hypothesis-django

hypothesis-django adds support for testing your Django models with Hypothesis. Using it is quite straightforward: All you need to do is subclass hypothesis.extra.django.TestCase or hypothesis.extra.django.TransactionTestCase and you can use @given as normal, and the transactions will be per example rather than per test function as they would be if you used @given with a normal django test suite (this is important because your test function will be called multiple times and you don’t want them to interfere with each other). Test cases on these classes that do not use @given will be run as normal.

I strongly recommend not using TransactionTestCase unless you really have to. Because Hypothesis runs this in a loop the performance problems it normally has are significantly exacerbated and your tests will be really slow.

In addition to the above, Hypothesis has some limited support for automatically deriving strategies for your model types, which you can then customize further.

Warning: Hypothesis creates saved models. This will run inside your testing transaction when using the test runner, but if you use the dev console this will leave debris in your database.

For example, using the trivial django project I have for testing:

```python
>>> from hypothesis.extra.django.models import models
>>> from toystore.models import Customer

>>> c = models(Customer).example()
>>> c
<Customer: Customer object>

>>> c.email
'jaime.urbina@gmail.com'

>>> c.name
'\U00109d3d\U000e07be\U000165f8\U0003fabf\U000c12cd\U000f1910\U00059f12\U000e07be\U000c12cd\U000e07be\U000519b0\U000165f8\U0003fabf\U0007bc31'

>>> c.age
-873375803
```

Hypothesis has just created this with whatever the relevant type of data is.

Obviously the customer’s age is implausible, so let’s fix that:

```python
>>> from hypothesis.strategies import integers

>>> c = models(Customer, age=integers(min_value=0, max_value=120)).example()

>>> c
<Customer: Customer object>

>>> c.age
5
```

You can use this to override any fields you like. Sometimes this will be mandatory: If you have a non-nullable field of a type Hypothesis doesn’t know how to create (e.g. a foreign key) then the models function will error unless you explicitly pass a strategy to use there.

You can also register a default strategy for a field type if you have custom one that Hypothesis doesn’t know about or want to override the normal behaviour for some reason:

```python
>>> from toystore.models import CustomishField, Customish

>>> models(Customish).example()  # hypothesis.errors.InvalidArgument: Missing arguments for mandatory field
```
customish for model Customish

```python
>>> from hypothesis.extra.django.models import add_default_field_mapping
>>> from hypothesis.strategies import just
>>> add_default_field_mapping(CustomishField, just("hi"))
>>> x = models(Customish).example()
>>> x.customish
'hi'
```

Note that this mapping is on exact type. Subtypes will not inherit it.
Stateful testing

Hypothesis offers support for a stateful style of test, where instead of trying to produce a single data value that causes a specific test to fail, it tries to generate a program that errors. In many ways, this sort of testing is to classical property based testing as property based testing is to normal example based testing.

The idea doesn’t originate with Hypothesis, though Hypothesis’s implementation and approach is mostly not based on an existing implementation and should be considered some mix of novel and independent reinventions.

This style of testing is useful both for programs which involve some sort of mutable state and for complex APIs where there’s no state per se but the actions you perform involve e.g. taking data from one function and feeding it into another.

The idea is that you teach Hypothesis how to interact with your program: Be it a server, a python API, whatever. All you need is to be able to answer the question “Given what I’ve done so far, what could I do now?”. After that, Hypothesis takes over and tries to find sequences of actions which cause a test failure.

Right now the stateful testing is a bit new and experimental and should be considered as a semi-public API: It may break between minor versions but won’t break between patch releases, and there are still some rough edges in the API that will need to be filed off.

This shouldn’t discourage you from using it. Although it’s not as robust as the rest of Hypothesis, it’s still pretty robust and more importantly is extremely powerful. I found a number of really subtle bugs in Hypothesis by turning the stateful testing onto a subset of the Hypothesis API, and you likely will find the same.

Enough preamble, lets see how to use it.

The first thing to note is that there are two levels of API: The low level but more flexible API and the higher level rule based API which is both easier to use and also produces a much better display of data due to its greater structure. We’ll start with the more structured one.

### 7.1 Rule based state machines

Rule based state machines are the ones you’re most likely to want to use. They’re significantly more user friendly and should be good enough for most things you’d want to do.

A rule based state machine is a collection of functions (possibly with side effects) which may depend on both values that Hypothesis can generate and also on values that have resulted from previous function calls.

You define a rule based state machine as follows:

```python
import unittest
from collections import namedtuple

from hypothesis.stateful import RuleBasedStateMachine, Bundle, rule
```
In this we declare a Bundle, which is a named collection of previously generated values. We define two rules which put data onto this bundle - one which just generates leaves with integer labels, the other of which takes two previously generated values and returns a new one.

We can then integrate this into our test suite by getting a unittest TestCase from it:

```python
TestTrees = BalancedTrees.TestCase
if __name__ == '__main__':
    unittest.main()
```

(these will also be picked up by py.test if you prefer to use that). Running this we get:

```
Step #1: v1 = leaf(x=0)
Step #2: v2 = split(left=v1, right=v1)
Step #3: v3 = split(left=v2, right=v1)
Step #4: check_balanced(tree=v3)
F
======================================================================
FAIL: runTest (hypothesis.stateful.BalancedTrees.TestCase)
======================================================================
Traceback (most recent call last):
  (...)
assert abs(self.size(tree.left) - self.size(tree.right)) <= 1
AssertionError
```

Note how it’s printed out a very short program that will demonstrate the problem.
...the problem of course being that we’ve not actually written any code to balance this tree at all, so of course it’s not balanced.

So let’s balance some trees.

```python
from collections import namedtuple

from hypothesis.stateful import RuleBasedStateMachine, Bundle, rule

Leaf = namedtuple('Leaf', ('label',))
Split = namedtuple('Split', ('left', 'right'))

class BalancedTrees(RuleBasedStateMachine):
    trees = Bundle('BinaryTree')
    balanced_trees = Bundle('balanced BinaryTree')

    @rule(target=trees, x=int)
    def leaf(self, x):
        return Leaf(x)

    @rule(target=trees, left=trees, right=trees)
    def split(self, left, right):
        return Split(left, right)

    @rule(tree=balanced_trees)
    def check_balanced(self, tree):
        if isinstance(tree, Leaf):
            return
        else:
            assert abs(self.size(tree.left) - self.size(tree.right)) <= 1, repr(tree)
            self.check_balanced(tree.left)
            self.check_balanced(tree.right)

    @rule(target=balanced_trees, tree=trees)
    def balance_tree(self, tree):
        return self.split_leaves(self.flatten(tree))

    def size(self, tree):
        if isinstance(tree, Leaf):
            return 1
        else:
            return self.size(tree.left) + self.size(tree.right)

    def flatten(self, tree):
        if isinstance(tree, Leaf):
            return (tree.label,)
        else:
            return self.flatten(tree.left) + self.flatten(tree.right)

    def split_leaves(self, leaves):
        assert leaves
        if len(leaves) == 1:
            return Leaf(leaves[0])
        else:
            mid = len(leaves) // 2
            return Split(
```
We’ve now written a really noddy tree balancing implementation. This takes trees and puts them into a new bundle of data, and we only assert that things in the balanced_trees bundle are actually balanced.

If you run this it will sit their silently for a while (you can turn on verbose output to get slightly more information about what’s happening. debug will give you all the intermediate programs being run) and then run, telling you your test has passed! Our balancing algorithm worked.

Now lets break it to make sure the test is still valid:

Changing the split to mid = max(len(leaves) // 3, 1) this should no longer balance, which gives us the following counter-example:

```python
v1 = leaf(x=0)
v2 = split(left=v1, right=v1)
v3 = balance_tree(tree=v1)
v4 = split(left=v2, right=v2)
v5 = balance_tree(tree=v4)
check_balanced(tree=v5)
```

Note that the example could be shrunk further by deleting v3. Due to some technical limitations, Hypothesis was unable to find that particular shrink. In general it’s rare for examples produced to be long, but they won’t always be minimal.

You can control the detailed behaviour with a Settings object on the TestCase (this is a normal hypothesis Settings object using the defaults at the time the TestCase class was first referenced). For example if you wanted to run fewer examples with larger programs you could change the settings to:

```python
TestTrees.settings.max_examples = 100
TestTrees.settings.stateful_step_count = 100
```

Which doubles the number of steps each program runs and halves the number of runs relative to the example. settings.timeout will also be respected as usual.

### 7.2 Generic state machines

The class GenericStateMachine is the underlying machinery of stateful testing in Hypothesis. In execution it looks much like the RuleBasedStateMachine but it allows the set of steps available to depend in essentially arbitrary ways on what has happened so far. For example, if you wanted to use Hypothesis to test a game, it could choose each step in the machine based on the game to date and the set of actions the game program is telling it it has available.

It essentially executes the following loop:

```python
machine = MyStateMachine()
try:
    for _ in range(n_steps):
        step = machine.steps().example()
        machine.execute_step(step)
finally:
    machine.teardown()
```

Where steps() and execute_step() are methods you must implement, and teardown is a method you can implement if you need to clean something up at the end. steps returns a strategy, which is allowed to depend arbitrarily on the current state of the test execution. Ideally a good steps implementation should be robust against minor changes in the
state. steps that change a lot between slightly different executions will tend to produce worse quality examples because they’re hard to simplify.

The steps method may depend on external state, but it’s not advisable and may produce flaky tests.

If any of execute_step or teardown produces an error, Hypothesis will try to find a minimal sequence of values steps such that the following throws an exception:

```python
try:
    machine = MyStateMachine()
    for step in steps:
        machine.execute_step(step)
finally:
    machine.teardown()
```

and such that at every point, the step executed is one that could plausible have come from a call to steps() in the current state.

Here’s an example of using stateful testing to test a broken implementation of a set in terms of a list (note that you could easily do something close to this example with the rule based testing instead, and probably should. This is mostly for illustration purposes):

```python
import unittest
from hypothesis.stateful import GenericStateMachine
from hypothesis import strategy
from hypothesis.specifiers import sampled_from, just

class BrokenSet(GenericStateMachine):
    def __init__(self):
        self.data = []

    def steps(self):
        add_strategy = strategy((just("add"), int))
        if not self.data:
            return add_strategy
        else:
            return (add_strategy |
                    strategy((just("delete"), sampled_from(self.data))))

    def execute_step(self, step):
        action, value = step
        if action == 'delete':
            try:
                self.data.remove(value)
            except ValueError:
                pass
            assert value not in self.data
        else:
            assert action == 'add'
            self.data.append(value)
            assert value in self.data

TestSet = BrokenSet.TestCase

if __name__ == '__main__':
```

7.2. Generic state machines
unittest.main()

Note that the strategy changes each time based on the data that’s currently in the state machine.

Running this gives us the following:

Step #1: ('add', 0)
Step #2: ('add', 0)
Step #3: ('delete', 0)
F

FAIL: runTest (hypothesis.stateful.BrokenSet.TestCase)

So it adds two elements, then deletes one, and throws an assertion when it finds out that this only deleted one of the copies of the element.
8 Compatibility

Hypothesis does its level best to be compatible with everything you could possibly need it to be compatible with. Generally you should just try it and expect it to work. If it doesn’t, you can be surprised and check this document for the details.

8.1 Python versions

Hypothesis has quite wide version support. It is supported and tested on python 2.7 and python 3.2+. Supporting 3.0 or 3.1 wouldn’t be infeasible but I’d need a good reason to. Supporting python before 2.7 isn’t going to happen.

Hypothesis also supports PyPy (PyPy3 should also work but isn’t part of the CI at the moment), and should support 32-bit and narrow builds, though this is currently only tested on Windows.

No testing has been performed on Jython or IronPython. It might work but I’d be surprised. Let me know if you need these supported. It might be possible but I make no promises.

8.2 Operating systems

In theory Hypothesis should work anywhere that Python does. In practice it is only known to work and regularly tested on OS X, Windows and Linux, and you may experience issues running it elsewhere. For example a known issue is that FreeBSD splits out the python-sqlite package from the main python package, and you will need to install that in order for it to work.

If you’re using something else and it doesn’t work, do get in touch and I’ll try to help, but unless you can come up with a way for me to run a CI server on that operating system it probably won’t stay fixed due to the inevitable march of time.

8.3 Testing frameworks

In general Hypothesis goes to quite a lot of effort to generate things that look like normal Python test functions that behave as closely to the originals as possible, so it should work sensibly out of the box with every test framework.

In terms of what’s actually known to work:

- Hypothesis integrates as smoothly with py.test and unittest as I can make it, and this is verified as part of the CI.
- Nose has been tried at least once and works fine, and I’m aware of people who use Hypothesis with Nose, but this isn’t tested as part of the CI.
• Integration with Django’s testing requires use of the `hypothesis-django` package. The issue is that in Django’s tests’ normal mode of execution it will reset the database one per test rather than once per example, which is not what you want.

Coverage works out of the box with Hypothesis (and Hypothesis has 100% branch coverage in its own tests). However you should probably not use Coverage, Hypothesis and PyPy together. Because Hypothesis does quite a lot of CPU heavy work compared to normal tests, it really exacerbates the performance problems the two normally have working together.

### 8.4 Regularly verifying this

Everything mentioned above as explicitly supported is checked on every commit with Travis and Appveyor and goes green before a release happens, so when I say they’re supported I really mean it.
This is a collection of examples of how to use Hypothesis in interesting ways. It’s small for now but will grow over time.

All of these examples are designed to be run under py.test (nose should probably work too).

### 9.1 How not to sort by a partial order

The following is an example that’s been extracted and simplified from a real bug that occurred in an earlier version of Hypothesis. The real bug was a lot harder to find.

Suppose we’ve got the following type:

```python
class Node(object):
    def __init__(self, label, value):
        self.label = label
        self.value = tuple(value)

    def __repr__(self):
        return "Node(%r, %r)" % (self.label, self.value)

    def sorts_before(self, other):
        if len(self.value) >= len(other.value):
            return False
        return other.value[:len(self.value)] == self.value
```

Each node is a label and a sequence of some data, and we have the relationship sorts_before meaning the data of the left is an initial segment of the right. So e.g. a node with value [1, 2] will sort before a node with value [1, 2, 3], but neither of [1, 2] nor [1, 3] will sort before the other.

We have a list of nodes, and we want to topologically sort them with respect to this ordering. That is, we want to arrange the list so that if x.sorts_before(y) then x appears earlier in the list than y. We naively think that the easiest way to do this is to extend the partial order defined here to a total order by breaking ties arbitrarily and then using a normal sorting algorithm. So we define the following code:

```python
from functools import total_ordering

@total_ordering
class TopoKey(object):
    def __init__(self, node):
        self.value = node
```
```python
def __lt__(self, other):
    if self.value.sorts_before(other.value):
        return True
    if other.value.sorts_before(self.value):
        return False
    return self.value.label < other.value.label

def sort_nodes(xs):
    xs.sort(key=TopoKey)
```

This takes the order defined by sorts_before and extends it by breaking ties by comparing the node labels.

But now we want to test that it works.

First we write a function to verify that our desired outcome holds:

```python
def is_prefix_sorted(xs):
    for i in range(len(xs)):
        for j in range(i+1, len(xs)):
            if xs[j].sorts_before(xs[i]):
                return False
    return True
```

This will return false if it ever finds a pair in the wrong order and return true otherwise.

Given this function, what we want to do with Hypothesis is assert that for all sequences of nodes, the result of calling sort_nodes on it is sorted.

First we need to define a strategy for Node:

```python
from hypothesis import Settings, strategy
import hypothesis.strategies as s

NodeStrategy = s.builds(
    Node,
    s.integers(),
    s.lists(s.booleans(), average_size=5, max_size=10))
```

We want to generate short lists of values so that there’s a decent chance of one being a prefix of the other (this is also why the choice of bool as the elements). We then define a strategy which builds a node out of an integer and one of those short lists of booleans.

We can now write a test:

```python
from hypothesis import given
@given(s.lists(Node))
def test_sorting_nodes_is_prefix_sorted(xs):
    sort_nodes(xs)
    assert is_prefix_sorted(xs)
```

this immediately fails with the following example:

```
[Node(0, (False, True)), Node(0, (True,)), Node(0, (False,))]
```

The reason for this is that because False is not a prefix of (True, True) nor vice versa, sorting things the first two nodes are equal because they have equal labels. This makes the whole order non-transitive and produces basically nonsense results.
But this is pretty unsatisfying. It only works because they have the same label. Perhaps we actually wanted our labels to be unique. Let’s change the test to do that.

```python
def deduplicate_nodes_by_label(nodes):
    table = {}
    for node in nodes:
        table[node.label] = node
    return list(table.values())
```

We define a function to deduplicate nodes by labels, and then map that over a strategy for lists of nodes to give us a strategy for lists of nodes with unique labels. We can now rewrite the test to use that:

```python
@given(NodeSet)
def test_sorting_nodes_is_prefix_sorted(xs):
    sort_nodes(xs)
    assert is_prefix_sorted(xs)
```

Hypothesis quickly gives us an example of this still being wrong:

```
[Node(0, (False,)), Node(-1, (True,)), Node(-2, (False, False))]
```

Now this is a more interesting example. None of the nodes will sort equal. What is happening here is that the first node is strictly less than the last node because (False,) is a prefix of (False, False). This is in turn strictly less than the middle node because neither is a prefix of the other and -2 < -1. The middle node is then less than the first node because -1 < 0.

So, convinced that our implementation is broken, we write a better one:

```python
def sort_nodes(xs):
    for i in hrange(1, len(xs)):
        j = i - 1
        while j >= 0:
            if xs[j].sorts_before(xs[j+1]):
                break
            xs[j], xs[j+1] = xs[j+1], xs[j]
            j -= 1
```

This is just insertion sort slightly modified - we swap a node backwards until swapping it further would violate the order constraints. The reason this works is because our order is a partial order already (this wouldn’t produce a valid result for a general topological sorting - you need the transitivity).

We now run our test again and it passes, telling us that this time we’ve successfully managed to sort some nodes without getting it completely wrong. Go us.

### 9.2 Time zone arithmetic

This is an example of some tests for pytz which check that various timezone conversions behave as you would expect them to. These tests should all pass, and are mostly a demonstration of some useful sorts of thing to test with Hypothesis, and how the hypothesis-datetime extra package works.

```python
from hypothesis import given, Settings
from hypothesis.extra.datetime import datetimes
from hypothesis.strategies import sampled_from
import pytz
from datetime import timedelta
```

9.2. Time zone arithmetic 45
ALL_TIMEZONES = list(map(pytz.timezone, pytz.all_timezones))

# There are a lot of fiddly edge cases in dates, so we run a larger number of
# examples just to be sure
with Settings(max_examples=1000):
  @given(
    datetimes(),  # datetimes generated are non-naive by default
    sampled_from(ALL_TIMEZONES), sampled_from(ALL_TIMEZONES),
  )
  def test_convert_via_intermediary(dt, tz1, tz2):
    """
    Test that converting between timezones is not affected by a detour via
    another timezone.
    """
    assert dt.astimezone(tz1).astimezone(tz2) == dt.astimezone(tz2)

  @given(
    datetimes(timezones=[]),  # Now generate naive datetimes
    sampled_from(ALL_TIMEZONES), sampled_from(ALL_TIMEZONES),
  )
  def test_convert_to_and_fro(dt, tz1, tz2):
    """
    If we convert to a new timezone and back to the old one this should
    leave the result unchanged.
    """
    dt = tz1.localize(dt)
    assert dt == dt.astimezone(tz2).astimezone(tz1)

  @given(
    datetimes(),
    sampled_from(ALL_TIMEZONES),
  )
  def test_adding_an_hour_commutes(dt, tz):
    """
    When converting between timezones it shouldn't matter if we add an hour
    here or add an hour there.
    """
    an_hour = timedelta(hours=1)
    assert (dt + an_hour).astimezone(tz) == dt.astimezone(tz) + an_hour

  @given(
    datetimes(),
    sampled_from(ALL_TIMEZONES),
  )
  def test_adding_a_day_commutes(dt, tz):
    """
    When converting between timezones it shouldn't matter if we add a day
    here or add a day there.
    """
    a_day = timedelta(days=1)
    assert (dt + a_day).astimezone(tz) == dt.astimezone(tz) + a_day
9.3 Condorcet’s Paradox

A classic paradox in voting theory, called Condorcet’s paradox, is that majority preferences are not transitive. That is, there is a population and a set of three candidates A, B and C such that the majority of the population prefer A to B, B to C and C to A.

Wouldn’t it be neat if we could use Hypothesis to provide an example of this?

Well as you can probably guess from the presence of this section, we can! This is slightly surprising because it’s not really obvious how we would generate an election given the types that Hypothesis knows about.

The trick here turns out to be twofold:

1. We can generate a type that is much larger than an election, extract an election out of that, and rely on minimization to throw away all the extraneous detail.

2. We can use assume and rely on Hypothesis’s adaptive exploration to focus on the examples that turn out to generate interesting elections

Without further ado, here is the code:

```python
from hypothesis import given, assume
from hypothesis.strategies import integers, lists
from collections import Counter

def candidates(votes):
    return {candidate for vote in votes for candidate in vote}

def build_election(votes):
    """
    Given a list of lists we extract an election out of this. We do this in two phases:

    1. First of all we work out the full set of candidates present in all votes and throw away any votes that do not have that whole set.
    2. We then take each vote and make it unique, keeping only the first instance of any candidate.

    This gives us a list of total orderings of some set. It will usually be a lot smaller than the starting list, but that's OK.
    """
    all_candidates = candidates(votes)
    votes = list(filter(lambda v: set(v) == all_candidates, votes))
    if not votes:
        return []
    rebuilt_voters = []
    for vote in votes:
        rv = []
        for v in vote:
            if v not in rv:
                rv.append(v)
        assert len(rv) == len(all_candidates)
        rebuilt_voters.append(rv)
    return rebuilt_voters

@given(lists(lists(integers(min_value=1, max_value=5))))
def test_elections_are_transitive(election):
```

9.3. Condorcet’s Paradox
Hypothesis Documentation, Release 1.5.0

```python
election = build_election(election)
# Small elections are unlikely to be interesting
assume(len(election) >= 3)
all_candidates = candidates(election)
# Elections with fewer than three candidates certainly can't exhibit
# intransitivity
assume(len(all_candidates) >= 3)

# Now we check if the election is transitive
# First calculate the pairwise counts of how many prefer each candidate
to the other

counts = Counter()
for vote in election:
    for i in range(len(vote)):
        for j in range(i + 1, len(vote)):
            counts[(vote[i], vote[j])] += 1

# Now look at which pairs of candidates one has a majority over the
# other and store that.

graph = {}
all_candidates = candidates(election)
for i in all_candidates:
    for j in all_candidates:
        if counts[(i, j)] > counts[(j, i)]:
            graph.setdefault(i, set()).add(j)

# Now for each triple assert that it is transitive.
for x in all_candidates:
    for y in graph.get(x, ()):  
        for z in graph.get(y, ()):
            assert x not in graph.get(z, ())
```

The example Hypothesis gives me on my first run (your mileage may of course vary) is:

```
[[3, 1, 4], [4, 3, 1], [1, 4, 3]]
```

Which does indeed do the job: The majority (votes 0 and 1) prefer 3 to 1, the majority (votes 0 and 2) prefer 1 to 4 and the majority (votes 1 and 2) prefer 4 to 3. This is in fact basically the canonical example of the voting paradox, modulo variations on the names of candidates.

### 9.4 Fuzzing an HTTP API

Hypothesis’s support for testing HTTP services is somewhat nascent. There are plans for some fully featured things around this, but right now they’re probably quite far down the line.

But you can do a lot yourself without any explicit support! Here’s a script I wrote to throw random data against the API for an entirely fictitious service called Waspfinder (this is only lightly obfuscated and you can easily figure out who I’m actually talking about, but I don’t want you to run this code and hammer their API without their permission).

All this does is use Hypothesis to generate random JSON data matching the format their API asks for and check for 500 errors. More advanced tests which then use the result and go on to do other things are definitely also possible.

```python
import unittest
from hypothesis import given, assume, Settings
from collections import namedtuple
import requests
```

Chapter 9. Some more examples
import os
import random
import time
import math
from hypothesis.strategies import one_of, sampled_from, lists

# These tests will be quite slow because we have to talk to an external
# service. Also we'll put in a sleep between calls so as to not hammer it.
# As a result we reduce the number of test cases and turn off the timeout.
Settings.default.max_examples = 100
Settings.default.timeout = -1

Goal = namedtuple("Goal", ("slug",))

# We just pass in our API credentials via environment variables.
waspfinder_token = os.getenv('WASPFINDER_TOKEN')
waspfinder_user = os.getenv('WASPFINDER_USER')
assert waspfinder_token is not None
assert waspfinder_user is not None

GoalData = {
    'title': str,
    'goal_type': sampled_from(lists
        "hustler", "biker", "gainer", "fatloser", "inboxer",
        "drinker", "custom"))
    'goaldate': one_of((None, float)),
    'goalval': one_of((None, float)),
    'rate': one_of((None, float)),
    'initval': float,
    'panic': float,
    'secret': bool,
    'datapublic': bool,
}

needs2 = ['goaldate', 'goalval', 'rate']

class WaspfinderTest(unittest.TestCase):
    @given(GoalData)
    def test_create_goal_dry_run(self, data):
        # We want slug to be unique for each run so that multiple test runs
        # don't interfere with eachother. If for some reason some slugs trigger
        # an error and others don't we'll get a Flaky error, but that's OK.
        slug = hex(random.getrandbits(32))[2:]

        # Use assume to guide us through validation we know about, otherwise
        # we'll spend a lot of time generating boring examples.

        # Title must not be empty
        assume(data["title"])

        # Exactly two of these values should be not None. The other will be
        # inferred by the API.
        assume(len([1 for k in needs2 if data[k] is not None]) == 2)
for v in data.values():
    if isinstance(v, float):
        assume(not math.isnan(v))
    data["slug"] = slug

    # The API nicely supports a dry run option, which means we don't have
    # to worry about the user account being spammed with lots of fake goals
    # Otherwise we would have to make sure we cleaned up after ourselves
    # in this test.
    data["dryrun"] = True
    data["auth_token"] = waspfinder_token
    for d, v in data.items():
        if v is None:
            data[d] = "null"
        else:
            data[d] = str(v)
    result = requests.post(  
        "https://waspfinder.example.com/api/v1/users/"  
        "%s/goals.json" % (waspfinder_user,), data=data)

    # Lets not hammer the API too badly. This will of course make the
    # tests even slower than they otherwise would have been, but that's
    # life.
    time.sleep(1.0)

    # For the moment all we're testing is that this doesn't generate an
    # internal error. If we didn't use the dry run option we could have
    # then tried doing more with the result, but this is a good start.
    self.assertNotEqual(result.status_code, 500)

if __name__ == '__main__':
    unittest.main()
Innovative features of Hypothesis

This document is a guide to Hypothesis internals, mostly with a goal to porting to other implementations of Quickcheck that want to benefit from some of the more unusual/interesting ideas in it, but it’s probably of general interest. It assumes you have some familiarity with the general ideas of property based testing and Quickcheck.

Nothing here is stable public API and might all be prone to change between minor releases. The purpose of this document is to share the ideas, not to specify the behaviour.

If you want to see all of these how most of these pieces fit together, there is also a worked example available here.

This is sorted roughly in order of most interesting to least technically interesting.

10.1 Templating

Templating is the single most important innovation in Hypothesis. If you’re going to take any ideas out of Hypothesis you should take this one.

The idea is as follows: Rather than generating data of the required type directly, value generation is split into two parts. We first generate a template and we then have a function which can reify that template, turning it into a value of the desired type. Importantly, simplification happens on templates and not on reified data.

This has several major advantages:

1. The templates can be of a much more restricted type than the desired output - you can require them to be immutable, serializable, hashable, etc without in any way restricting the range of data that you can generate.

2. Seamless support for mutable data: Because the mutable object you produce is the result of reifying the template, any mutation done by the function you call does not affect the underlying template.

3. Generation strategies are monads (more or less. The generation is monadic, the simplification rules don’t strictly follow the monad laws but this isn’t a problem in practice).

The latter is worth elaborating on: Hypothesis SearchStrategy has methods map and flatmap, which lets you do e.g. strategy(int).map(lambda x: Decimal(x) / 100).

This gives you a new strategy for decimals, which still supports minimization. The normal obstacle here is that you can’t minimize the result because you’d need a way to map back to the original data type, but that isn’t an issue here because you can just keep using the previous template type, minimize that, and only convert to the new data type at the point of reification.

Making generation monadic is trickier because of the way it has to interact with reification (you can’t know what the strategy you need to draw from is until you’ve reified the intermediate argument, which you can’t do). The way this is solved is pretty fiddly and involves some tricks that wouldn’t work in a pure language unless reify was also pure (and it’s quite useful to allow reify to be monadic).
10.2 Multi-stage simplification

Hypothesis generally seems to try harder than classic quickcheck to produce simple examples. Unfortunately this meant historically that simplification was potentially very slow. Multi-stage simplification helps with this a lot by avoiding large categories of behaviours that waste time.

The core idea is that there are different categories of simplification, and once a category of simplification has stopped working you should stop trying it even if you’ve changed other things. For example, if we have something like:

```python
@given([int])
def test_lists_are_short(xs):
    assert len(xs) < 100
```

then in the classic mode of quickcheck simplification, once we’ve found an example which is only 100 elements long and are trying to simplify the elements to find out if they are essential, each recursive simplification will nevertheless try to shrink the size of the list, wasting a lot of time after each successful shrink of an element.

The way Hypothesis solves this is to split simplification into stages: Instead of a single function simplify, we have a list (well, generator) of simplify functions.

This gives us the following algorithm (somewhere between python and pseudocode):

```python
def minimize_with_shrinker(x, f, shrinker):
    ""
    Greedily apply a single shrinker function to find a smaller version 
of x which satisfies f.
    ""
    for s in shrinker(x):
        if f(s):
            return minimize_with_shrinker(s, f, shrinker)
    return x

def shrink_pass(x, f):
    ""
    Apply each shrinker in turn to minimizing an example from x 
    ""
    for shrinker in shrinkers:
        x = minimize_with_shrinker(x, f, shrinker)
    return x

def minimize(x, f):
    ""
    Repeatedly do minimization passes on x until we hit a fixed point 
    ""
    while True:
        shrunk = shrink_pass(x, f)
        if shrunk == x:
            return shrunk
        x = shrunk
```

So in the list example we have two simplification passes: The first attempts to remove elements, the second attempts to simplify elements in place without changing the size of the list.

We do multiple passes because sometimes a later pass can unblock a condition that was making a previous pass make progress by e.g. changing relations between elements.

In order to avoid combinatorial explosions when recursively applying simplification one will frequently flatten down the simplification passes for elements into a single pass, using the function
def all_shrinks(x):
    shrink in shrinkers:
        for s in shrink(x):
            yield s

Empirically this general approach seems to be much faster for classes of example where one of the passes is constrained, while still producing high quality results.

An additional detail: In actual fact, the function that returns the shrinkers has access to the value to be shrunk. This is to handle the case where there might be a very large number of potential shrinkers, most of them useless. In the monadic case we have an infinite space of potential shrinkers because we can only apply shrinkers from the target strategy if we know the source value.

The shrink functions returned must all be able to handle any value (in the sense of not erroring. They don’t have to do anything useful). The purpose of the argument to shrinkers is only to immediately eliminate shrinkers that won’t be useful.

### 10.3 Parametrization

Template generation is also less direct than you might expect. Each strategy has two distributions: A parameter distribution, and a conditional template distribution given a parameter value.

The idea is that a parameter value says roughly what sort of things should be generated, and then the template distribution generates them given that specification.

To consider a simple example, a parameter value for a generating booleans is a number between 0 and 1 which is the probability of generating true. So in order to draw a boolean we draw that number from a uniform distribution, then we draw a boolean which is true with that probability.

As described, the result is indistinguishable from just flipping a coin. The resulting bool will be true 50% of the time. The interesting thing is how parameters compose.

Suppose we now want to draw a list of booleans. This will have a parameter value which is a pair of numbers: The first is the expected length, the second is the bool parameter, which is the probability of any given element being true.

This allows us to reach a lot of values that would be essentially impossible to reach otherwise. Suppose we needed a list of length at least 20 elements all of which are true in order to trigger a bug. Given a length of 20, if each element is drawn independently the chances of them all being true are just under one in a million. However with this parametrization it’s one in 21 (because if you draw a number close to 1 it makes them all more likely to be true).

The idea of trying to generate this sort of “clumpier” distribution is based on a paper called Swarm Testing, but with some extensions to the idea. The essential concept is that a distribution which is too flat is likely to spend too much time exploring uninteresting interactions. By making any given draw focus on some particular area of the search space we significantly increase the chances of certain interesting classes of things happening.

The second important benefit of the parameter system is that you can use it to guide the search space. This is useful because it allows you to use otherwise quite hard to satisfy preconditions in your tests.

The way this works is that we store all the parameter values we’ve used, and will tend to use each parameter value multiple times. Values which tend to produce “bad” results (that is, produce a test such that assume() is called with a falsey value and rejects the example it was given) will be chosen less often than a parameter value which doesn’t. Values which produce templates we’ve already seen are also penalized in order to guide the search towards novelty.

The way this works in Hypothesis is with an infinitely many armed bandit algorithm based on Thompson Sampling and some ad hoc hacks I found useful to avoid certain pathological behaviours. I don’t strongly recommend following the specific algorithm, though it seems to work well in practice, but if you want to take a look at the code it’s in this file.
10.4 The database

There’s not much to say here except “why isn’t everyone doing this?” (though in fairness this is made much easier by the template system).

When Hypothesis finds a minimal failing example it saves the template for it in a database (by default a local sqlite database, though it could be anything). When run in future, Hypothesis first checks if there are any saved examples for the test and tries those first. If any of them fail the test, it skips straight to the minimization stage without bothering with data generation. This is particularly useful for tests with a low probability of failure - if Hypothesis has a one in 1000 chance of finding an example it will probably take 5 runs of the test suite before the test fails, but after that it will consistently fail until you fix the bug.

The key that Hypothesis uses for this is the type signature of the test, but that hasn’t proven terribly useful. You could use the name of the test equally well without losing much.

I had some experiments with disassembling and reassembling examples for reuse in other tests, but in the end these didn’t prove very useful and were hard to support after some other changes to the system, so I took them out.

A minor detail that’s worth bearing in mind: Because the template type of a strategy is not considered part of its public API, it may change in a way that makes old serialized data in the database invalid. Hypothesis handles this in a “self-healing” way by validating the template as it comes out of the database and silently discarding any that don’t correspond to a valid template.

10.5 Example tracking

The idea of this is simply that we don’t want to call a test function with the same example twice. I think normal property based testing systems don’t do this because they just assume that properties are faster to check than it is to test whether we’ve seen this one before, especially given a low duplication rate.

Because Hypothesis is designed around the assumption that you’re going to use it on things that look more like unit tests (and also because Python is quite slow) it’s more important that we don’t duplicate effort, so we track which templates have previously been run and don’t bother to reify and test them again if they come up. As mentioned in the previous section we also then penalize the parameter that produced them.

This is also useful for minimization: Hypothesis doesn’t mind if you have cycles in your minimize graph (e.g. if x simplifies to y and y simplifies to x) because it can just use the example tracking system to break loops.

There’s a trick to this: Examples might be quite large and we don’t actually want to keep them around in memory if we don’t have to. Because of the restricted templates, we can insist that all examples belong to a set of types that have a stable serialization format. So rather than storing and testing the whole examples for equality we simply serialize them and (if the serialized string is at least 20 bytes) we take the sha1 hash of it. We then just keep these hashes around and if we’ve seen the hash before we treat the example as seen.

10.6 The strategy function

Hypothesis uses an extensible function called strategy that basically means “convert this object into a strategy if it’s not one already”. This turns out to be a really good API for quickcheck style things in a dynamic language, because it means you can very often do “things that look like types” to map to a strategy, and it also lets you do nice things like putting in custom strategies anywhere you want.

I only mention this because I spent a lot of time with a much worse API and it looks like this is not something that has generally been settled on very clearly for dynamic languages. I believe the more common approach is to just use combinators for everything, but the Hypothesis one looks a lot prettier.
This is a record of all past Hypothesis releases and what went into them, in reverse chronological order. All previous releases should still be available on pip.

Hypothesis APIs come in three flavours:

- Public: Hypothesis releases since 1.0 are semantically versioned with respect to these parts of the API. These will not break except between major version bumps. All APIs mentioned in this documentation are public unless explicitly noted otherwise.
- Semi-public: These are APIs that are considered ready to use but are not wholly nailed down yet. They will not break in patch releases and will usually not break in minor releases, but when necessary minor releases may break semi-public APIs.
- Internal: These may break at any time and you really should not use them at all.

You should generally assume that an API is internal unless you have specific information to the contrary.

11.1 1.4.0 - 2015-05-04

Codename: What a state.

The big feature of this release is the new and slightly experimental stateful testing API. You can read more about that in the appropriate section.

Two minor features the were driven out in the course of developing this:

- You can now set settings.max_shrinks to limit the number of times Hypothesis will try to shrink arguments to your test. If this is set to <= 0 then Hypothesis will not rerun your test and will just raise the failure directly. Note that due to technical limitations if max_shrinks is <= 0 then Hypothesis will print every example it calls your test with rather than just the failing one. Note also that I don’t consider settings max_shrinks to zero a sensible way to run your tests and it should really be considered a debug feature.
- There is a new debug level of verbosity which is even more verbose than verbose. You probably don’t want this.

Breakage of semi-public SearchStrategy API:

- It is now a required invariant of SearchStrategy that if u simplifies to v then it is not the case that strictly_simpler(u, v). i.e. simplifying should not increase the complexity even though it is not required to decrease it. Enforcing this invariant lead to finding some bugs where simplifying of integers, floats and sets was suboptimal.
- Integers in basic data are now required to fit into 64 bits. As a result python integer types are now serialized as strings, and some types have stopped using quite so needlessly large random seeds.
Hypothesis Stateful testing was then turned upon Hypothesis itself, which lead to an amazing number of minor bugs being found in Hypothesis itself.

Bugs fixed (most but not all from the result of stateful testing) include:

- Serialization of streaming examples was flaky in a way that you would probably never notice: If you generate a template, simplify it, serialize it, deserialize it, serialize it again and then deserialize it you would get the original stream instead of the simplified one.

- If you reduced max_examples below the number of examples already saved in the database, you would have got a ValueError. Additionally, if you had more than max_examples in the database all of them would have been considered.

- @given will no longer count duplicate examples (which it never called your function with) towards max_examples. This may result in your tests running slower, but that’s probably just because they’re trying more examples.

- General improvements to example search which should result in better performance and higher quality examples. In particular parameters which have a history of producing useless results will be more aggressively culled. This is useful both because it decreases the chance of useless examples and also because it’s much faster to not check parameters which we were unlikely to ever pick!

- integers_from and lists of types with only one value (e.g. [None]) would previously have had a very high duplication rate so you were probably only getting a handful of examples. They now have a much lower duplication rate, as well as the improvements to search making this less of a problem in the first place.

- You would sometimes see simplification taking significantly longer than your defined timeout. This would happen because timeout was only being checked after each successful simplification, so if Hypothesis was spending a lot of time unsuccessfully simplifying things it wouldn’t stop in time. The timeout is now applied for unsuccessful simplifications too.

- In Python 2.7, integers_from strategies would have failed during simplification with an OverflowError if their starting point was at or near to the maximum size of a 64-bit integer.

- flatmap and map would have failed if called with a function without a __name__ attribute.

- If max_examples was less than min_satisfying_examples this would always error. Now min_satisfying_examples is capped to max_examples. Note that if you have assumptions to satisfy here this will still cause an error.

Some minor quality improvements:

- Lists of streams, flatmapped strategies and basic strategies should now now have slightly better simplification.

### 11.2 1.3.0 - 2015-04-22

New features:

- New verbosity level API for printing intermediate results and exceptions.

- New specifier for strings generated from a specified alphabet.

- Better error messages for tests that are failing because of a lack of enough examples.

Bug fixes:

- Fix error where use of ForkingTestCase would sometimes result in too many open files.

- Fix error where saving a failing example that used flatmap could error.

- Implement simplification for sampled_from, which apparently never supported it previously. Oops.
General improvements:

- Better range of examples when using one_of or sampled_from.
- Fix some pathological performance issues when simplifying lists of complex values.
- Fix some pathological performance issues when simplifying examples that require unicode strings with high codepoints.
- Random will now simplify to more readable examples.

11.3 1.2.1 - 2015-04-16

A small patch release for a bug in the new executors feature. Tests which require doing something to their result in order to fail would have instead reported as flaky.

11.4 1.2.0 - 2015-04-15

Codename: Finders keepers.

A bunch of new features and improvements.

- Provide a mechanism for customizing how your tests are executed.
- Provide a test runner that forks before running each example. This allows better support for testing native code which might trigger a segfault or a C level assertion failure.
- Support for using Hypothesis to find examples directly rather than as just as a test runner.
- New streaming type which lets you generate infinite lazily loaded streams of data - perfect for if you need a number of examples but don’t know how many.
- Better support for large integer ranges. You can now use integers_in_range with ranges of basically any size. Previously large ranges would have eaten up all your memory and taken forever.
- Integers produce a wider range of data than before - previously they would only rarely produce integers which didn’t fit into a machine word. Now it’s much more common. This percolates to other numeric types which build on integers.
- Better validation of arguments to @given. Some situations that would previously have caused silently wrong behaviour will now raise an error.
- Include +/- sys.float_info.max in the set of floating point edge cases that Hypothesis specifically tries.
- Fix some bugs in floating point ranges which happen when given +/- sys.float_info.max as one of the endpoints... (really any two floats that are sufficiently far apart so that x, y are finite but y - x is infinite). This would have resulted in generating infinite values instead of ones inside the range.

11.5 1.1.1 - 2015-04-07

Codename: Nothing to see here

This is just a patch release put out because it fixed some internal bugs that would block the Django integration release but did not actually affect anything anyone could previously have been using. It also contained a minor quality fix for floats that I’d happened to have finished in time.
• Fix some internal bugs with object lifecycle management that were impossible to hit with the previously released versions but broke hypothesis-django.
• Bias floating point numbers somewhat less aggressively towards very small numbers

### 11.6 1.1.0 - 2015-04-06

Codename: No-one mention the M word.

• Unicode strings are more strongly biased towards ascii characters. Previously they would generate all over the space. This is mostly so that people who try to shape their unicode strings with assume() have less of a bad time.
• A number of fixes to data deserialization code that could theoretically have caused mysterious bugs when using an old version of a Hypothesis example database with a newer version. To the best of my knowledge a change that could have triggered this bug has never actually been seen in the wild. Certainly no-one ever reported a bug of this nature.
• Out of the box support for Decimal and Fraction.
• New dictionary specifier for dictionaries with variable keys.
• Significantly faster and higher quality simplification, especially for collections of data.
• New filter() and flatmap() methods on Strategy for better ways of building strategies out of other strategies.
• New BasicStrategy class which allows you to define your own strategies from scratch without needing an existing matching strategy or being exposed to the full horror or non-public nature of the SearchStrategy interface.

### 11.7 1.0.0 - 2015-03-27

Codename: Blast-off!

There are no code changes in this release. This is precisely the 0.9.2 release with some updated documentation.

### 11.8 0.9.2 - 2015-03-26

Codename: T-1 days.

• floats_in_range would not actually have produced floats_in_range unless that range happened to be (0, 1). Fix this.

### 11.9 0.9.1 - 2015-03-25

Codename: T-2 days.

• Fix a bug where if you defined a strategy using map on a lambda then the results would not be saved in the database.
• Significant performance improvements when simplifying examples using lists, strings or bounded integer ranges.
11.10 0.9.0 - 2015-03-23

Codename: The final countdown

This release could also be called 1.0-RC1.

It contains a teeny tiny bugfix, but the real point of this release is to declare feature freeze. There will be zero functionality changes between 0.9.0 and 1.0 unless something goes really really wrong. No new features will be added, no breaking API changes will occur, etc. This is the final shakedown before I declare Hypothesis stable and ready to use and throw a party to celebrate.

Bug bounty for any bugs found between now and 1.0: I will buy you a drink (alcoholic, caffeinated, or otherwise) and shake your hand should we ever find ourselves in the same city at the same time.

The one tiny bugfix:

• Under pypy, databases would fail to close correctly when garbage collected, leading to a memory leak and a confusing error message if you were repeatedly creating databases and not closing them. It is very unlikely you were doing this and the chances of you ever having noticed this bug are very low.

11.11 0.7.2 - 2015-03-22

Codename: Hygienic macros or bust

• You can now name an argument to @given ‘f’ and it won’t break (issue #38)
• strategy_test_suite is now named strategy_test_suite as the documentation claims and not in fact strategy_test_suitee
• Settings objects can now be used as a context manager to temporarily override the default values inside their context.

11.12 0.7.1 - 2015-03-21

Codename: Point releases go faster

• Better string generation by parametrizing by a limited alphabet
• Faster string simplification - previously if simplifying a string with high range unicode characters it would try every unicode character smaller than that. This was pretty pointless. Now it stops after it’s a short range (it can still reach smaller ones through recursive calls because of other simplifying operations).
• Faster list simplification by first trying a binary chop down the middle
• Simultaneous simplification of identical elements in a list. So if a bug only trickers when you have duplicates but you drew e.g. [-17, -17], this will now simplify to [0, 0].

11.13 0.7.0 - 2015-03-20

Codename: Starting to look suspiciously real

This is probably the last minor release prior to 1.0. It consists of stability improvements, a few usability things designed to make Hypothesis easier to try out, and filing off some final rough edges from the API.

• Significant speed and memory usage improvements
• Add an example() method to strategy objects to give an example of the sort of data that the strategy generates.
• Remove .descriptor attribute of strategies
• Rename descriptor_test_suite to strategy_test_suite
• Rename the few remaining uses of descriptor to specifier (descriptor already has a defined meaning in Python)

11.14 0.6.0 - 2015-03-13

Codename: I'm sorry, were you using that API?

This is primarily a “simplify all the weird bits of the API” release. As a result there are a lot of breaking changes. If you just use @given with core types then you're probably fine.

In particular:

• Stateful testing has been removed from the API
• The way the database is used has been rendered less useful (sorry). The feature for reassembling values saved from other tests doesn’t currently work. This will probably be brought back in post 1.0.
• SpecificationMapper is no longer a thing. Instead there is an ExtMethod called strategy which you extend to specify how to convert other types to strategies.
• Settings are now extensible so you can add your own for configuring a strategy
• MappedSearchStrategy no longer needs an unpack method
• Basically all the SearchStrategy internals have changed massively. If you implemented SearchStrategy directly rather than using MappedSearchStrategy talk to me about fixing it.
• Change to the way extra packages work. You now specify the package. This must have a load() method. Additionally any modules in the package will be loaded in under hypothesis.extra

Bug fixes:

• Fix for a bug where calling falsify on a lambda with a non-ascii character in its body would error.

Hypothesis Extra:

hypothesis-fakefactory: An extension for using faker data in hypothesis. Depends on fake-factory.

11.15 0.5.0 - 2015-02-10

Codename: Read all about it.

Core hypothesis:

• Add support back in for pypy and python 3.2
• @given functions can now be invoked with some arguments explicitly provided. If all arguments that hypothesis would have provided are passed in then no falsification is run.
• Related to the above, this means that you can now use pytest fixtures and mark.parametrize with Hypothesis without either interfering with the other.
• Breaking change: @given no longer works for functions with varargs (varkwargs are fine). This might be added back in at a later date.
• Windows is now fully supported. A limited version (just the tests with none of the extras) of the test suite is run on windows with each commit so it is now a first class citizen of the Hypothesis world.
• Fix a bug for fuzzy equality of equal complex numbers with different reprs (this can happen when one coordinate is zero). This shouldn’t affect users - that feature isn’t used anywhere public facing.

• Fix generation of floats on windows and 32-bit builds of python. I was using some struct.pack logic that only worked on certain word sizes.

• When a test times out and hasn’t produced enough examples this now raises a Timeout subclass of Unfalsifiable.

• Small search spaces are better supported. Previously something like a @given(bool, bool) would have failed because it couldn’t find enough examples. Hypothesis is now aware of the fact that these are small search spaces and will not error in this case.

• Improvements to parameter search in the case of hard to satisfy assume. Hypothesis will now spend less time exploring parameters that are unlikely to provide anything useful.

• Increase chance of generating “nasty” floats

• Fix a bug that would have caused unicode warnings if you had a sampled_from that was mixing unicode and byte strings.

• Added a standard test suite that you can use to validate a custom strategy you’ve defined is working correctly.

Hypothesis extra:

First off, introducing Hypothesis extra packages!

These are packages that are separated out from core Hypothesis because they have one or more dependencies. Every hypothesis-extra package is pinned to a specific point release of Hypothesis and will have some version requirements on its dependency. They use entry_points so you will usually not need to explicitly import them, just have them installed on the path.

This release introduces two of them:

hypothesis-datetime:

Does what it says on the tin: Generates datetimes for Hypothesis. Just install the package and datetime support will start working.

Depends on pytz for timezone support

hypothesis-pytest:

A very rudimentary pytest plugin. All it does right now is hook the display of falsifying examples into pytest reporting.

Depends on pytest.

11.16 0.4.3 - 2015-02-05

Codename: TIL narrow Python builds are a thing

This just fixes the one bug.

• Apparently there is such a thing as a “narrow python build” and OSX ships with these by default for python 2.7. These are builds where you only have two bytes worth of unicode. As a result, generating unicode was completely broken on OSX. Fix this by only generating unicode codepoints in the range supported by the system.

11.17 0.4.2 - 2015-02-04

Codename: O(dear)

This is purely a bugfix release:
• Provide sensible external hashing for all core types. This will significantly improve performance of tracking seen examples which happens in literally every falsification run. For Hypothesis fixing this cut 40% off the runtime of the test suite. The behaviour is quadratic in the number of examples so if you’re running the default configuration this will be less extreme (Hypothesis’s test suite runs at a higher number of examples than default), but you should still see a significant improvement.

• Fix a bug in formatting of complex numbers where the string could get incorrectly truncated.

### 11.18 0.4.1 - 2015-02-03

Codename: Cruel and unusual edge cases

This release is mostly about better test case generation.

Enhancements:

• Has a cool release name

• text_type (str in python 3, unicode in python 2) example generation now actually produces interesting unicode instead of boring ascii strings.

• floating point numbers are generated over a much wider range, with particular attention paid to generating nasty numbers - nan, infinity, large and small values, etc.

• examples can be generated using pieces of examples previously saved in the database. This allows interesting behaviour that has previously been discovered to be propagated to other examples.

• improved parameter exploration algorithm which should allow it to more reliably hit interesting edge cases.

• Timeout can now be disabled entirely by setting it to any value <= 0.

Bug fixes:

• The descriptor on a OneOfStrategy could be wrong if you had descriptors which were equal but should not be coalesced. e.g. a strategy for one_of((frozenset({int}), {int})) would have reported its descriptor as {int}. This is unlikely to have caused you any problems

• If you had strategies that could produce NaN (which float previously couldn’t but e.g. a Just(float(‘nan’)) could) then this would have sent hypothesis into an infinite loop that would have only been terminated when it hit the timeout.

• Given elements that can take a long time to minimize, minimization of floats or tuples could be quadratic or worse in the that value. You should now see much better performance for simplification, albeit at some cost in quality.

Other:

• A lot of internals have been been rewritten. This shouldn’t affect you at all, but it opens the way for certain of hypothesis’s oddities to be a lot more extensible by users. Whether this is a good thing may be up for debate...

### 11.19 0.4.0 - 2015-01-21

FLAGSHIP FEATURE: Hypothesis now persists examples for later use. It stores data in a local SQLite database and will reuse it for all tests of the same type.

LICENSING CHANGE: Hypothesis is now released under the Mozilla Public License 2.0. This applies to all versions from 0.4.0 onwards until further notice. The previous license remains applicable to all code prior to 0.4.0.

Enhancements:
• Printing of failing examples. I was finding that the pytest runner was not doing a good job of displaying these, and that Hypothesis itself could do much better.

• Drop dependency on six for cross-version compatibility. It was easy enough to write the shim for the small set of features that we care about and this lets us avoid a moderately complex dependency.

• Some improvements to statistical distribution of selecting from small (<= 3 elements)

• Improvements to parameter selection for finding examples.

Bugs fixed:

• could_have_produced for lists, dicts and other collections would not have examined the elements and thus when using a union of different types of list this could result in Hypothesis getting confused and passing a value to the wrong strategy. This could potentially result in exceptions being thrown from within simplification.

• sampled_from would not work correctly on a single element list.

• Hypothesis could get very confused by values which are equal despite having different types being used in descriptors. Hypothesis now has its own more specific version of equality it uses for descriptors and tracking. It is always more fine grained than Python equality: Things considered != are not considered equal by hypothesis, but some things that are considered == are distinguished. If your test suite uses both frozenset and set tests this bug is probably affecting you.

11.20 0.3.2 - 2015-01-16

• Fix a bug where if you specified floats_in_range with integer arguments Hypothesis would error in example simplification.

• Improve the statistical distribution of the floats you get for the floats_in_range strategy. I’m not sure whether this will affect users in practice but it took my tests for various conditions from flaky to rock solid so it at the very least improves discovery of the artificial cases I’m looking for.

• Improved repr() for strategies and RandomWithSeed instances.

• Add detection for flaky test cases where hypothesis managed to find an example which breaks it but on the final invocation of the test it does not raise an error. This will typically happen with too much recursion errors but could conceivably happen in other circumstances too.

• Provide a “derandomized” mode. This allows you to run hypothesis with zero real randomization, making your build nice and deterministic. The tests run with a seed calculated from the function they’re testing so you should still get a good distribution of test cases.

• Add a mechanism for more conveniently defining tests which just sample from some collection.

• Fix for a really subtle bug deep in the internals of the strategy table. In some circumstances if you were to define instance strategies for both a parent class and one or more of its subclasses you would under some circumstances get the strategy for the wrong superclass of an instance. It is very unlikely anyone has ever encountered this in the wild, but it is conceivably possible given that a mix of namedtuple and tuple are used fairly extensively inside hypothesis which do exhibit this pattern of strategy.

11.21 0.3.1 - 2015-01-13

• Support for generation of frozenset and Random values

• Correct handling of the case where a called function mutates it argument. This involved introducing a notion of a strategies knowing how to copy their argument. The default method should be entirely acceptable and the
worst case is that it will continue to have the old behaviour if you don’t mark your strategy as mutable, so this shouldn’t break anything.

- Fix for a bug where some strategies did not correctly implement could_have_produced. It is very unlikely that any of these would have been seen in the wild, and the consequences if they had been would have been minor.
- Re-export the @given decorator from the main hypothesis namespace. It’s still available at the old location too.
- Minor performance optimisation for simplifying long lists.

11.22 0.3.0 - 2015-01-12

- Complete redesign of the data generation system. Extreme breaking change for anyone who was previously writing their own SearchStrategy implementations. These will not work any more and you’ll need to modify them.
- New settings system allowing more global and modular control of Verifier behaviour.
- Decouple SearchStrategy from the StrategyTable. This leads to much more composable code which is a lot easier to understand.
- A significant amount of internal API renaming and moving. This may also break your code.
- Expanded available descriptors, allowing for generating integers or floats in a specific range.
- Significantly more robust. A very large number of small bug fixes, none of which anyone is likely to have ever noticed.
- Deprecation of support for pypy and python 3 prior to 3.3. 3.3 and 3.4. Supported versions are 2.7.x, 3.3.x, 3.4.x. I expect all of these to remain officially supported for a very long time. I would not be surprised to add pypy support back in later but I’m not going to do so until I know someone cares about it. In the meantime it will probably still work.

11.23 0.2.2 - 2015-01-08

- Fix an embarrassing complete failure of the installer caused by my being bad at version control

11.24 0.2.1 - 2015-01-07

- Fix a bug in the new stateful testing feature where you could make __init__ a @requires method. Simplification would not always work if the prune method was able to successfully shrink the test.

11.25 0.2.0 - 2015-01-07

- It’s aliivve.
- Improve python 3 support using six.
- Distinguish between byte and unicode types.
- Fix issues where FloatStrategy could raise.
- Allow stateful testing to request constructor args.
• Fix for issue where test annotations would timeout based on when the module was loaded instead of when the test started

11.26 0.1.4 - 2013-12-14

• Make verification runs time bounded with a configurable timeout

11.27 0.1.3 - 2013-05-03

• Bugfix: Stateful testing behaved incorrectly with subclassing.
• Complex number support
• Support for recursive strategies
• Different error for hypotheses with unsatisfiable assumptions

11.28 0.1.2 - 2013-03-24

• Bugfix: Stateful testing was not minimizing correctly and could throw exceptions.
• Better support for recursive strategies.
• Support for named tuples.
• Much faster integer generation.

11.29 0.1.1 - 2013-03-24

• Python 3.x support via 2to3.
• Use new style classes (oops).

11.30 0.1.0 - 2013-03-23

• Introduce stateful testing.
• Massive rewrite of internals to add flags and strategies.

11.31 0.0.5 - 2013-03-13

• No changes except trying to fix packaging

11.32 0.0.4 - 2013-03-13

• No changes except that I checked in a failing test case for 0.0.3 so had to replace the release. Doh
11.33 0.0.3 - 2013-03-13

- Improved a few internals.
- Opened up creating generators from instances as a general API.
- Test integration.

11.34 0.0.2 - 2013-03-12

- Starting to tighten up on the internals.
- Change API to allow more flexibility in configuration.
- More testing.

11.35 0.0.1 - 2013-03-10

- Initial release.
- Basic working prototype. Demonstrates idea, probably shouldn’t be used.
Contributing

External contributions to Hypothesis are currently less easy than I would like them to be. You might want to consider any of the following in preference to trying to work on the main Hypothesis code base:

- Submit bug reports
- Submit feature requests
- Write about Hypothesis
- Build libraries and tools on top of Hypothesis outside the main repo

And indeed I’ll be delighted with you if you do! If you need any help with any of these, get in touch and I’ll be extremely happy to provide it.

However if you really really want to submit code to Hypothesis, the process is as follows:

You must own the copyright to the patch you’re submitting as an individual. I’m not currently clear on how to accept patches from organisations and other legal entities.

If you have not already done so, you must sign a CLA assigning copyright to me. Send an email to hypothesismaciver.com with an attached copy of the current version of the CLA and the text in the body “I, (your name), have read the attached CLA and agree to its terms” (you should in fact have actually read it).

Note that it’s important to attach a copy of the CLA because I may change it from time to time as new things come up and this keeps a record of which version of it you agreed to.

Then submit a pull request on Github. This will be checked by Travis and Appveyor to see if the build passes.

Advance warning that passing the build requires:

1. All the tests to pass, naturally.
2. Your code to have 100% branch coverage.
3. Your code to be flake8 clean.
4. Your code to be a fixed point for a variety of reformattting operations (defined in lint.sh)

It is a fairly strict process.

Once all this has happened I’ll review your patch. I don’t promise to accept it, but I do promise to review it as promptly as I can and to tell you why if I reject it.
12.1 Documentation changes

The CLA is still required for significant documentation changes, but if you want to just submit typo fixes and so on I’ll happily just merge them. A CLA request for fixing a typo seems a bit silly.
h

hypothesis.strategies, 21
Index

B
basic() (in module hypothesis.strategies), 22
basic_simplify() (hypothesis.strategies.SearchStrategy method), 29
binary() (in module hypothesis.strategies), 22
booleans() (in module hypothesis.strategies), 21
builds() (in module hypothesis.strategies), 22
C
complex_numbers() (in module hypothesis.strategies), 21
D
decimals() (in module hypothesis.strategies), 22
dictionaries() (in module hypothesis.strategies), 22
draw_parameter() (hypothesis.strategies.SearchStrategy method), 28
draw_template() (hypothesis.strategies.SearchStrategy method), 28
E
elementary() (hypothesis.strategies.SearchStrategy method), 27
F
filter() (hypothesis.strategies.SearchStrategy method), 28
fixed_dictionaries() (in module hypothesis.strategies), 22
flatmap() (hypothesis.strategies.SearchStrategy method), 28
floats() (in module hypothesis.strategies), 21
fractions() (in module hypothesis.strategies), 22
from_basic() (hypothesis.strategies.SearchStrategy method), 28
frozensets() (in module hypothesis.strategies), 22
full_simplify() (hypothesis.strategies.SearchStrategy method), 29
H
hypothesis.strategies (module), 21
I
integers() (in module hypothesis.strategies), 21
J
just() (in module hypothesis.strategies), 21
L
lists() (in module hypothesis.strategies), 21
M
map() (hypothesis.strategies.SearchStrategy method), 28
O
one_of() (in module hypothesis.strategies), 21
R
reify() (hypothesis.strategies.SearchStrategy method), 28
S
SearchStrategy (class in hypothesis.strategies), 27
sets() (in module hypothesis.strategies), 22
simplifiers() (hypothesis.strategies.SearchStrategy method), 28
streaming() (in module hypothesis.strategies), 22
strictly_simpler() (hypothesis.strategies.SearchStrategy method), 28
T
template_upper_bound (hypothesis.strategies.SearchStrategy attribute), 28
text() (in module hypothesis.strategies), 22
to_basic() (hypothesis.strategies.SearchStrategy method), 28
tuples() (in module hypothesis.strategies), 21